2004

An implementation of feasible path constraints generation for reproducible testing.

Jun Li
University of Windsor

An Implementation of Feasible Path Constraints Generation for Reproducible Testing

By

Jun Li

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada
March, 2004

Your file Votre référence
ISBN: 0-612-92498-X
Our file Notre référence
ISBN: 0-612-92498-X

Abstract

Non-determinism features make the testing of a concurrent program not repeatable. Specification-based *reproducible testing* is a promising technique that may give the tester more control over the environment of concurrent testing. With a given test case, the crucial part of the *test scenario* which contributes to achieving the control on the execution path are input events and *path constraints* in terms of synchronization events. The problem considered in this thesis is to generate a significant set of *path constraints* automatically from the design specification in terms of *design abstract* under the assumption that monitors are the key mechanism to handle the synchronization events. In addition, as a considerable feature, *formal methods* have been applied in the implementation tool to construct the *path constraints*.

Keywords: *Non-determinism, Reproducible testing, Path constraints, Formal methods, Structural Operational Semantics, Labeled transition systems, Automata theory.*

Acknowledgement

I would like to take this opportunity to thank my supervisor, Dr. Jessica Chen, for her invaluable guidance and advices, for her enthusiastic encouragement and her great patience to me. I specially thank my committee members, Dr. Christie Ezeife, Dr Zhiguo Hu and Dr. Joan Morrissey for spending their precious time to read this thesis and for their suggestions on the thesis work. I would also like to thank my wife, Xiying Sheng, for her understanding, patience and support, and thank my whole family, and my friends, especially Mrs. Madeleine Godwin, for supporting me in so many ways during my study period at the University of Windsor.

Contents

List of Tables

List of Figures

Chapter 1 Motivation

With the growing complexity of computation, concurrent software systems are becoming more prevalent. While concurrent programming has brought with it the power of processing multiple tasks simultaneously, it also has presented a lot of challenges to software developers. One particular challenge is testing.

Software testing is the process of checking the functionality of system implementations by making them execute under certain conditions. Such a process is usually performed either to detect possible deficiencies or to measure the reliability of the given implementations. Testing plays a vital role in the procedure of software development, and is an important means for us to gain confidence in the quality of a software product. A bundle of well-defined techniques has been widely used for developing traditional sequential programs. However, it is far from simple to deal with the concurrent systems. The major difficulty comes from the characteristic of non-determinism that is inherited by all concurrent programs. In particular, while it can be guaranteed that with fixed input the same output will be produced in the sequential programs, users of concurrent programs may get different results in every run, even with the exact same feed-in data. One reason for such an outcome is the existence of *race condition* that is introduced by the unpredictable executing rates of different processes. The concept of a process is analogous to a thread in a multithreaded system. Since the speeds of these processes are not determined, multiple executions of a concurrent program may exercise different sequences of visiting shared variables and exchanging messages. Consequently, this feature of non-determinism makes the testing of a concurrent program non-repeatable. In other words, the tester of concurrent programs may not have the chance to observe a certain erroneous phenomenon that occurred before. In addition, the tested program may exercise some identical path many times or never exercise some other feasible paths.

1.1 Techniques for concurrent system testing

To deal with the problem of non-determinism, a lot of techniques that intend to enforce the under-testing concurrent program to follow certain execution paths artificially in order to reproduce the same testing phenomenon were proposed in recent years [3, 5, 10, 11, 13, 32, 41, 42]. These techniques generally fall into two categories: debugging-related techniques and specification-based testing techniques.

1.1.1 Debugging-related testing techniques

To achieve the control of deterministic execution over non-deterministic programs, special debugging techniques can be applied.

Software debugging is a process different from testing. It aims at locating the causes of some observed errors in a software program. Debugging a sequential program usually involves multiple iterations of the executions of a program, and on purpose, pause the execution at certain well-chosen points according to the user's experience, in order to examine the current state. Similarly, a primitive way to debug a concurrent system is to re-execute the traced computation in a controlled fashion. However, manually inserting the control mechanism is not an efficient solution due to the extreme complexity in most of the concurrent systems. Therefore, rather than using traditional debugging tools, some other more sophisticated techniques were introduced in the past decade. For instance, a predicate control mechanism that allows computations to be run based on added synchronizations is presented by [36]. The idea of predicate control is to manage the program execution by artificially inserting synchronizations that do not violate the functionalities of the underlying program. With such predicates, particular executing orders can be ensured so that the goal of replaying can be achieved. An event-based approach for debugging is reported in [3]. This approach accomplishes the debugging by creating models of desired program behaviours and comparing these models with the actual behaviour of under-debugging systems.

1.1.2 Specification-based testing techniques

Generally, software testing may be divided into two basic approaches: program-based testing and specification-based testing.

Testing based on program code itself is conducted in an intuitive way, and studies of such approaches, which are performed to testing concurrent programs, can be found in [14, 26, 44, 52, 57].

Specification-based testing is a totally different case. We need to gather the test input from the specification of a software system. As the foundation for testing, these specifications must be complete, precise and unambiguous, otherwise misinterpretation may occur and even a disaster of testing failure can possibly take place. Furthermore, these specifications must contain the correctness criterion that describes the expected system behaviours under every possible circumstance, where system behaviours are any observable activities during a system execution. One of the advantages brought by specification-based testing is that we are able to perform tests before the implementation is finished; possible deficiencies of specification may be found early, and test data can be generated independently from the concrete implementation. As a consequence, the test cost will be significantly reduced in terms of time and money.

Some discussions on specification-based testing have been investigated in [1, 40]. More specifically, to test a concurrent program, instead of simply using the model such as finite state machine, certain control or guidance is also needed [10, 11, 13, 41, 42].

1.2 Our problem

Due to the non-determinism of concurrent systems, testing is not simple to perform. Specification-based *reproducible testing* is a promising technique that employs a set of predefined control points that can be used to automatically handle the order of executions of each process. To gain desired control for *reproducible testing*, certain information called a *test scenario* has to be provided. Usually, a *test scenario* must include not only a *test case* but also a *path constraint* for this *test case*. For the sake of simplicity, we only consider non-distributed concurrent systems in this thesis work. Thus, the *test case* refers to only a sequence of inputs and expected outputs. In the present work, the *test case* is

assumed to be given. On the other hand, the *path constraint,* which is the crucial part of a *test scenario* and contributes to achieving the control on the execution path, can be expressed as an order among some specific internal events.

Before introducing *path constraints*, a set of interested events has to be pre-defined. Since the prominent task of testing is to consider possible deficiencies of the underlying program, the *path constraints* are very often designed to disclose the representative scenarios that may likely contain the bugs or errors. Due to the characteristics of the concurrent system, the interested events will be focused on those related to synchronization activities, for instance, accessing shared objects and coordinating between different process, and more detailed discussion can be found in [11].

In fact, it is observed that these synchronization activities represent the origin or cause of non-deterministic behaviour such that different outputs are produced with the same inputs. There are a number of mechanisms available for accomplishing the synchronization, some well-known ones of which are *busy-waiting protocols*, *semaphore* and *monitor* [2]. Busy-waiting is an implementation of synchronization in which a process repeatedly examines a certain condition until it becomes true. The downsides of using busy-waiting protocols include: most of such protocols are quite complex; there lacks clear distinction between variables that are employed for synchronization and those that are used for program computation; and it is inefficient to apply the busy-waiting in most concurrent programs. A semaphore is a special kind of shared variable that allows only two atomic operations, p and v. The atomic operation refers to a unitary operation that is essentially indivisible and unchangeable. The p operation is applied to ensure a process can proceed only if an event has occurred, while such event occurrences can be signaled by perform the v operation. However, semaphores are also a low-level mechanism and may introduce some errors when it is being applied. A monitor is a program module where at most one client may execute a routine of it at any given time. In this thesis work, the monitor is considered as the key mechanism to handle the synchronization mechanisms since it is more structured and efficient than the others, and is made available in a variety of concurrent programming languages.

Another important factor that may yield notable influence over the synchronization events is the timing of the input, since it is most likely in concurrent programs a sequence

of input will be used to supply more than one process. Thus, to achieve the control over execution of concurrent programs, we must be able to determine the time of occurrences of those events including not only synchronization events but also input events. Therefore, each *path constraint* is a sequence of synchronization events and input events, which corresponds to the control points in the PUT. Finally, the control over program execution can be obtained by adjusting the order of these control points.

One testing approach concerned with how such a control mechanism forces the Program Under Test (PUT) to execute exactly as the desired path according to the given input and *path constraints* was discussed in [10]. In particular, such control is accomplished by placing the PUT into a well-designed testing environment, suspending the execution at certain control points, and enabling message exchanging between the control mechanism and all the processes in the PUT in order to determine if the specific process should continue or wait until some specific events occur. These control points are the moments that occur immediately prior to or subsequent to the synchronization events, and the moments that occur immediately prior to or subsequent to the input events. Other approaches dealt with controlling such forced execution of concurrent programs that, via managing the run-time scheduler or debugger, can be found in [11, 32].

However, generating *path constraints* is difficult, expensive, and tedious. First of all, the specification of a software system may be imprecise, incomplete, and inconsistent. Second, very often, there are lack of some efficient and effective ways for deriving a control model in which contains the information of all *path constraints* from the given design specification. Finally, since such control models are usually huge, an appropriate method is desired to simplifying them without loosing any necessary information.

1.3 Contribution of Thesis Work

This thesis work involves a framework for automatically generating significant sets of feasible *path constraints* for *reproducible testing* from the design specification and the given *test case*.

As a considerable feature, *formal methods* will be applied to construct the *path constraints*. *Formal methods* refer to mathematical-based techniques that can be applied to specify, develop and verify not only software but also hardware systems, and will be further discussed in ***Chapter 3.1***. Also, by assumption, we consider the PUT of a static set of processes and the design abstract of the PUT is provided in terms of formal specification language *process terms* [35], where the details of *process terms* will be given in ***Chapter 3.4***. Formal modeling language *Labelled Transition System (LTS)* is explored; and Structural Operational Semantics is used to systematically and automatically produce such an *LTS* which include necessary information to retrieve all feasible *path constraints* (a detailed introduction to *LTS* can be found in ***Chapter 3.2***). Trace equivalence, which is a kind of equivalence relation between different states in the program computation, will be chosen to simplify the *labeled transition system* by ignoring irrelevant internal events and reserving only those labels of the synchronization events and input events, and the trace *equivalence is* going to be introduced in ***Chapter 3.3***.

By surveying various related works, critical reviews and comparisons between those techniques and the present work have been made by this thesis. Based on the *process terms* and *LTS* discussed in the previous researches of Dr. Jessica Chen, an algorithm for generating an *LTS* model from the given design abstract in terms of *process terms* has been provided in this thesis. Meanwhile, the algorithms for simplifying such an *LTS* have been constructed according to the theory of *Finite Automata*. Besides, this thesis also gives the algorithm of deriving the desired path constraints for gaining control of reproducible testing. With these algorithms, a tool for automatically generating significant sets of feasible *path constraints* has been implemented. Afterwards, experiments have been done, and the results of which proves the feasibility and efficiency of the proposed framework.

1.4 Structure of the thesis

The remainder of the thesis is organized as follows: ***Chapter 2*** gives the overview and comparison of related works. ***Chapter 3*** introduces the background of the techniques

and some previous work that are going to be used in the implementation. ***Chapter 4*** illustrates the generation of *LTS* by an example application: *producer and consumer problem*. ***Chapter 5*** discusses the implementation details of the proposed framework. ***Chapter 6*** displays the evaluation of this implementation. ***Chapter 7*** provides a conclusion and offers a discussion of future work.

Chapter 2 Related works

This chapter reviews some of the important related works and approaches that concern the problems with *reproducible testing*. Studies on testing concurrent systems have drawn more and more attention from researchers during the past few years. Most of their work has been focused on monitoring or controlling the execution of the system in terms of the nondeterministic behavior [3, 5, 8, 10, 11, 13, 28, 31, 32, 41, 42].

Reproducible testing is an effective technique to enable deterministic testing for concurrent programs, which allows a specific test scenario to be replayed [41, 42, 43]. Usually, two important issues are considered while performing the *reproducible testing*: generation of the *test scenario* and realization of the desired execution, the focus of this thesis work will be limited to the former.

The traditional replay control techniques used to be the hot spot of concurrent program testing, and some of them have been introduced in [11, 32, 36]. With these techniques, a certain mechanism will be applied to record the internal choices that are related to the nondeterministic behavior of the PUT when the PUT is running with some inputs. Afterwards, the replay control mechanism can be used to force the PUT to execute according to the recorded choices. Such replay techniques are crucial for *regression testing* which is a testing that intends to re-test the unmodified functionalities in case some corrections or modifications for the PUT have been made. In contrast, *reproducible testing* does not necessarily need the PUT to be executed first in order to record the relevant messages to construct the controlled execution sequence. In other words, such an execution sequence can be acquired from a number of other sources such as requirement documents, design documents, and program codes (except from recording in the previous execution of PUT).

It is commonly believed by most researchers that both test case and the sequence of execution of events, in terms of statements, are needed to be taken into account while performing *reproducible testing*. More specifically, the considerations of such execution events are focused on the sequences of concurrency-related statements [3, 5, 10, 11, 13,

32] and remote method invocation involved statements [8, 10, 41, 43]. However, since distributed concurrent systems are beyond the scope of the present research, the sequences of statements will be concerned only with respect to concurrency control in this thesis work.

Current approaches suggested by researches in the area of reproducible testing include *test case* generation for traditional systems [20, 38, 56], *CSPE*-based testing [13], state-based testing for CORBA applications [41], techniques for integrating *formal method* into *reproducible testing* [8], applications of *labeled transition system* for concurrent systems [49, 51].

2.1 Test case generation for traditional systems

Creating *test cases* is laborious, high-priced, and annoying. The traditional way is to build an automatic generating tool. Generally, there are two kinds of approaches for generating *test cases*: *code-based* and *specification-based.*

Code-based test case generation derives *test cases* from the actual code. Some of the methodologies that are used with code-based testing include *statement coverage* that requires all the statements in the PUT to be covered at least once, and *branch coverage* that requires all the branches of conditional statements to be covered at least once. One classical example of the code-based tool for generating *test cases* is a tool called Godzilla [15, 16]. Although code-based *test cases* are effective due to the fact that they concern the way the software is actually written, code-based *test case* generation has a major disadvantage: the tests are based on the real implementation which may not be coherent with the specified requirements.

On the other hand, *specification-based test case* generation extracts test case based on the specification of what the software is supposed to do. Since *specification-based* testing only considers an external view of the software, it is not necessary to cover all of the statements of the PUT. Approaches of specification-based test case generation often fall into three groups: *model-based, algebraic* and *finite state machines-based.* The *finite state machine (*also known as *finite state automaton)* is a computational model consisting of a set of states which include a start state, an input alphabet, and a transition function

which maps input symbols and current states to some succeeding states. To provide some more robust test case generation methodologies, combinations of these techniques are most likely applied simultaneously. For instance, approaches introduced in [20, 56] have used *algebraic specifications*, *model-based specifications*, and *finite state machines*. A method introduced in [56] was intended to derive a *Finite State Model* that can be used to control the test process from specifications which are written in *Z* language. An approach to class-level test case generation from formal *object-oriented* specifications has been discussed in [20]. This approach first extracts a *test model*, which is a representation containing all the information to generate test cases from the design specifications. Such specifications are written in a language which includes an algebraic specification that consists of a number of functional modules and a set of object information that identifies the class name, invariants, historical constraints, and methods of each class. Thus, the test cases can be selected based on the partitioned input space from the *test model*. Another example concerns about generating test case automatically from design specification was described in [38]. This generating method is based on a formal specification language called Structured Object-Oriented Formal Language (SOFL).

However, the techniques and approaches illustrated above concerned only *test cases* for traditional non-concurrent systems, whereas the present work considers the testing over concurrent systems. Meanwhile, in the present work, the *test case* is assumed to be given instead of being generated from specification.

2.2 CSPE-based testing

A specification-based methodology which was designed for the purpose of testing concurrent programs based on sequencing constraints named *CSPE* was presented in [13].

Events of synchronization (or *SYN-events*) and the sequence of such events (or *SYN-sequence*) were thought to be the key when analyzing the behavior of concurrent programs. The feasibility and validity of a SYN-sequence depends on the acceptance from the implementation and specification of the underlying program, respectively. Thus, a *synchronization fault* is defined as either a feasible *SYN-sequence* of program with a

certain input considered to be invalid or a valid *SYN-sequence* of program with a certain input considered to be infeasible. In order to specify the sequencing constraints which are in harmony with the feasible *SYN-sequences* of a concurrent program, *Constraints on Succeeding and Preceding Events (CSPE)* has been defined. A strategy of generating such *CSPE* constraints automatically was proposed in [7, 12, 31]. In order to achieve more flexibility and expressiveness, these *CSPE* constraints is further abstracted by using strategies of equivalence, and only those observable events of program execution will be considered. Hence, the testing methodology proposed in [13] can be described as follows: first, derive a set of validity constraints in terms of *CSPE* constraints from specification of PUT; second, execute the PUT repeatedly with same input in order to collect the exercised SYN-sequence, by which coverage can be measured and violations of PUT's validity constraints can be detected; third, the deterministic testing, which involves forcing the PUT to execute with a specific input in harmony with a SYN-sequence, can be performed with the above-generated SYN-sequence; finally, possible constraint violations can be exposed if such deterministic testing cannot cover a constraint.

In essence, *CSPE*-based testing is a testing based on the specification in terms of Finite State Machine (*FSM*). Usually, the test generation criterion for an *FSM* makes use of *transition coverage* which requires every transition in the FSM to be covered at least once. However, *transition coverage* is not strong enough to detect certain error states. Therefore, instead of using *transition coverage,* the *CSPE-1*coveragae criterion which requires each constraint to be covered at least once is employed for *CSPE*-based test generation.

Based on such *CSPE* constraints, the sequence of the test can be either produced manually, or derived automatically from system specifications that are modeled with *FSM*. To automatically generate the test sequence, a strategy was presented in [31]. In particular, a weighted directed tree representation called a *constraint tree* is used in generating the test sequence. In the *constraint tree*, each node represents a constraint, and it is referred to as *valid* or *invalid* according to the validity of its labeled constraint. Directed edges between nodes denote the order of the corresponding constraint events. Hence, any path on the constraint tree indicates a test sequence. The step after the generation of the *constraint tree* is to select a minimum set of test sequences, in which

the calculation upon every possible combination of nodes will be performed. This guarantees that every constraint is covered at least once in a test sequence. By marking all the nodes in each combination and getting the maximum path which includes all the marked nodes and ends with a marked node, a set of *test sequences* which all conform to the *CSPE-1* coverage criterion will be produced.

There are a number of common features between such *CSPE*-based testing and the present work. For example, both are specification-based and both take into account the synchronization events. However, significant differences also exist. Comparatively, the *Labeled Transition System* is used as the means to generate the *path constraints* in the present work while the *constraint tree* is employed to derive the *test sequences* in the work of [31]. Meanwhile, while the testers have to specify the restrictions on the allowed sequences of synchronization events with the testing methodology based on the use of *CSPE* constraints, the *path constraints* in the present work are derived directly from the given design abstract by generating a *Labeled Transition System* that contains all possible serialization of the synchronization events. Furthermore, applying the *CSPE-1* coverage criterion in *CSPE*-based testing is due to the fact that a *CSPE* constraint contains only a temporal property and therefore lacks an efficient method to discover all possible serializations of the synchronization events that satisfy the given set of constraints. In the present work, some general coverage criteria such as *state coverage criterion* or *edge coverage criterion* can be applied to generate possible paths.

2.3 State-based testing for CORBA applications

Unlike the control structure based testing that was discussed in the last section, a *state-based reproducible testing* described in [30, 41, 42, 43] is capable of handling the complexity caused by the introduction of object-oriented structure and middleware technologies in the component-based software.

A *state machine model* which is based on the formalism of *statechart* is presented as the basis of the approach in [41]. The characteristics of *statechart* benefit this *state machine model* to deal with the concurrent, hierarchical and communicating problems of component-based distributed systems. One advantage of using *statechart* formalism is a

hierarchical feature allowing a set of states, which has the same meaning, to be replaced by one new state with some related transitions that are possibly guarded by certain conditions. Therefore, the state number in the new derived *state machine* will be reduced dramatically. Essentially, during the concurrent execution of component-based software, each method in the components may non-deterministically alter the state of the system. There are two types of *state machines* that are used to model the behavior of the PUT. The first one is the *atomic state machine* (*ASM*) which is employed to describe the state behavior for a single shared variable. The second type is the *composite state machine* (*CSM*) that is used to characterize the situation whereby a program involves more than one shared variable. For the state dependent behavior of a concurrent CORBA (Common Object Request Broker Architecture) implementation, such a set of *finite state machines* is defined for a set of interesting shared variables which are conveyed either in IDL or in the global declaration part.

When the construction of a *state machine* (*CSM*) for modeling the PUT is completed, a set of test sequences can be generated by building the test tree [30, 43]. To generate the test tree, the set of initial states that comes from each *ASM* in the *CSM* is used as the root node of the test tree. From the root node, a number of branches can be added according to all the alternative transitions that are all valid outgoing transitions from the root. Afterwards, a replay mechanism is used for a selected test sequence of a CORBA implementation. This replay mechanism is designed to perform the deterministic execution of a CORBA program in order to test such program based on a specific expected state behavior of the program. Since the PUT is actually a distributed concurrent program, the generation of the alternation of remote method invocation has also been considered.

With such a state-based testing approach, a state behavior error of a component-based program can be examined dynamically and deterministically. However, this approach did not take *formal methods* into account, whereas the introduction of *formal methods* is one of the prominent features of the present work. Meanwhile, this approach did not consider the different serializations of program execution based on synchronization events. As for the distributed concurrent systems, this testing approach

concerned remote method invocation activities in the mechanism of replay while the distributed characteristics are not considered in the present work.

2.4 Integration of formal method with reproducible testing

As mentioned in an earlier context, *formal methods* will be introduced as an important feature in the *present work.* In fact, the integration of *formal methods* with reproducible testing has also been explored by [8].

The PUTs considered in [8] are those middleware-based distributed concurrent systems. As discussed in ***Section 2.3,*** unlike testing a non-distributed concurrent program where all the processes reside locally, the remote method calls of a distributed system bring the extra challenge to the control mechanism of testing. For instance, the middleware CORBA may use one of the following thread models to manage the incoming method call inside the server: (1) a specific thread will be created to deal with each single remote call; (2) a specific thread will be created to deal with a number of remote calls on one particular object; and (3) a pool of threads will be created to deal with all the incoming method calls. In this case, traditional test control techniques which only focus on the synchronization matters are not able to handle, and may even add the new deadlock into the execution of the PUT. Therefore, according to the features of the distributed system, not only the order of synchronization events and input events but also the order of remote method calls has to be taken into account for the test control mechanism.

A static analysis technique was proposed in [8] to construct a *test model* in terms of finite automata for the distributed concurrent *PUTs*. Such a *test model* considers two kinds of events: synchronization events and remote call events. For each kind of event, both request points and completing points will be examined. In particular, an event is represented in this *test model* by a 7-tuple which includes the information about the originate process, target process, the object on which the calling method resides, the type of the event, and so on. The test constraint is expressed by the happen-before relation, for instance, e_1 -> e_2, where e_1, e_2 represent events. It is assumed that the test constraint which concerns these synchronization events and remote method call events is given in a

formal presentation. Meanwhile, this given formal presentation provides a binary relation which describes not only the test constraints but also the relationship between each synchronization event and its corresponding remote method call events in case that CORBA middleware is used for the process communications, and a function for determining the maximum number of threads in the thread pool of a specific process if the third thread model is used. Based on this information, *finite automaton* that contains all the possible execution paths of *PUT* can be defined. Since such automation constructed from test constraints most likely contains certain deadlock states, an algorithm is also given to derive the above automation to a new deadlock-free automation called *test model* by removing those states that may lead to the deadlock state. Thus, such a *test model* can be used by the test controller in reproducible testing for middleware-based *PUT*s to make a decision on whether or not to allow a request for remote method invocation or for shared object accessing. Moreover, this *test model* guarantees the test procedure will never introduce any new deadlock state.

The techniques presented in [8] appear in a lot of places similar to the present work. First of all, both of them utilize the *formal methods*. Second, constructing the *test model* in [8] and generating the control model in present work are all by performing the static analysis based on some given information, for instance, *design abstracts* in the present work. Finally, they both consider the synchronization events as the interested events. On the other hand, the major difference between them is that the main purpose of the *test model* in [8] is to force the PUT to execute according to the given test constraints (or *path constraints*) and guarantee that no new deadlock can be introduced; whereas the present work considers the generation of *path constraints*.

2.5 Applications of labeled transition system for concurrent systems

As mentioned in the first Chapter and ***Section 2.4***, a *labeled transition system (LTS)* will be employed in the *present work* as the major means to generate the *path constraint*. In fact, *LTS* has been applied as a well-defined model for concurrent systems over the past 20 years. Meanwhile, a great deal of formal literatures has taken into account the use of *LTS* to conduct testing; also, an annotated bibliography was presented in [6]

Conformance testing involves systematically testing the behavior of a software system based on its specification while not having the knowledge of its internal structure. Traditionally, *conformance testing* is also called *black box testing* or functional testing. One example to demonstrate the integration of *conformance testing* and *LTS* is given in [49, 51]. The ideas and experiences to support such integration are also presented in [25, 46, 47, 50]. On the other hand, the testing theory of *LTS*s was not originated with the system specification before. In fact, testing with *LTS*s used to be involved with modeling implementations to a transition system and determining the equivalence between the constructed model and the original implementation by examining whether the observation made by testing such a model with a set of given *test case*s is the same as the observation made by testing the real implementation. To fill such a gap, a framework for using formal methods in *conformance testing* was presented in [27, 49, 52]. A variety of concepts used in the procedure of formal *conformance testing* were provided at a high level abstraction in this framework. At the same time, a formal structure was defined in such a framework that reasoning about the testing became possible. Essentially, such a framework enhanced the formalism of the testing process, and bridged the informal part of testing such as implementations with the formal part including specification and models. With this framework, the implementation relation is defined by using such an observational framework and instantiating it with *LTS*. Consequently, the functional behavior of an implementation can be tested with regard to a formal specification. For instance, an *ioco testing,* which stands for input/output *conformance testing,* discussed in [46] requires the specifications to be given in terms of *LTS*s or other formal language with *LTS* semantics. Meanwhile, it is assumed that the implementations can be modeled by so-called *input-output transition systems* in which the idea was inherited from Input/Output Automata [33]. One test derivation algorithm introduced in [49, 51] was designed to derive the *test case*s from such formal specifications. The soundness of these derived *test cases* has been discussed in [48].

The *LTSs* in above-mentioned works are applied to express the allowed behavior of the system with possible inputs and outputs. Correspondingly, the labels in a *LTS* are grouped into two categories: labels concerned with inputs and labels concerned with outputs. In this case, one classical assumption about complete testing, which considers all

possible execution paths to a specific *test case* in the implementation will be exercised after performing the testing with such a *test case* a certain number of times, are often applied. However, the situation is different in the present work since certain control mechanisms are used to gain control over the internal choice with *reproducible testing* rather than relying on the complete testing assumption. Furthermore, the *LTS* is employed in the present work to describe the allowed behavior with given input in order to derive the *test model* which contains all the *path constraints* instead of generating test cases.

Chapter 3 Background and previous works

As mentioned in the first Chapter of this thesis, a variety of techniques are employed in the implementation for constructing the *path constraints*. Such techniques include *formal methods*, *Labelled Transition System (LTS)*, and *trace equivalence*, all of which are presented in detail in this Chapter. Furthermore, when applying such background knowledge to the work of implementing a tool for generating *path constraints* automatically, it is important to introduce some previous works that have been done by Dr. Jessica Chen [9]. These works include facilitating a specific format of process terms that will be used to construct the design abstract of the Program Under Test (PUT), giving a case of applying *LTS*, and defining a number of rules for generating the *LTS*.

3.1 Formal Methods

Applying *formal methods* is one of the prominent features of this implementation. Since mathematics has been introduced as the major feature of *formal methods*, these methods are empowered to handle the complexity of various modelling tasks [25, 27, 45, 50, 51].

Traditionally, specification of a software system is written in some natural languages such as the English language. As mentioned earlier in this Chapter, the specification can be used as the basis of testing. However, problems such as impreciseness, incompleteness, inconsistency and ambiguity of specification, which are caused by either human error or lack of experience, may occur frequently. These problems can not only impose such difficulties as being unable to determine the objective of underlying testing while generating the test case, but also render problematic task of analyzing the test result in terms of uncertainty of some particular issues, for example, suspected errors. Applying the formal methods to system specification brings an opportunity to figure out the above-mentioned problems. In a manner, the testing

requirements and design intentions in a specification will no longer require abstruse interpretation since the formal expressions are precise and consistent enough to be understood without equivocality by both software developers and software testers.

Furthermore, the application of *formal methods* enables the specification to be conveyed in a more detailed manner. Since the preciseness and completeness of specification are ensured by using *formal methods*, the possible independent decisions made by programmers can be greatly minimized. Therefore, a common problem during the software development procedure that the implementations may be improper, inadequate and not harmonious with the original purpose of the designer can be solved. Due to such a fact, right after the process of software design in terms of specification is finishing, the activity of testing can start immediately at the same time of programming instead of being delayed until the actual implementation is completed. Consequently, possible ambiguity, inconsistency and incompleteness can be found early during the development process which is one of major benefits of applying the *formal methods* and can greatly lower the developing cost.

Another fascinating advantage of applying *formal methods* is that the automation of testing can be accomplished. Testing in many cases is not simple due to the excessive complexity of real world applications. It is not surprising that testing may become a laborious, time-consuming and error-prone process in most situations. In fact, testing in the developing cycle always consumes a major portion of the funding. A sound solution to accommodate this problem is to introduce the automation into testing. It is not hard to imagine that by making the testing process automatic, the efficiency either on the issue of speed or resource consuming will highly enhanced. On the other hand, the error caused by human imperfection can be mainly eliminated by performing the testing routine automatically. Moreover, the testing process will become more reproducible if it can be executed without human interference and interpretation. Since the preciseness, completeness and unambiguity of specification by using *formal methods* can be guaranteed, and formal language instead of natural language is employed to express the specification, the specification is qualified to be a good basis of testing and manageable by well-defined tools. As a consequence, more automation of testing can be brought out.

Some good experiences of successfully using *formal methods* in software engineering are reported by [18, 19, 24, 29].

In recent years, *formal methods* were adopted more and more in software engineering. PROMELA is a formal language that is frequently used for communication protocol modelling which was introduced in [22]. Usually, PROMELA is working with SPIN as its input language and the latter serves as a model-checking tool for the formal verification of distributed systems. Another important formal language is Z which is based on *Zermelo-Fraenkel set theory* and *first order predicate logic* [39]. These two languages are employed in the work of [18, 29] to describe the formal specifications.

Other *formal methods* have been developed including those for SDL (Specification and Description Language) [36], for Abstract Data Type specification [17], for FSM (Finite State Machines) [34], and for LOTOS (Language of Temporal Ordering Specifications) [4]. In this thesis work, the *formal methods* of choice are *process algebra* and *Labeled Transition System* which is going to be discussed in detail later on (see ***Section 3.2***).

3.2 Labelled transition system

As one of the *formal methods*, the *labelled transition system* is an important modeling language [35, 46, 47, 53], and it is also used as the basic semantics for LOTOS.

A *labelled transition system* (*LTS*) is a quadruple <State, Label,→, s_0>, where

- State is a set of states during the execution of the process;
- Label is a set of labels displaying the information about the state conversion;
- → ⊆ State × Label × State is a set of transitions that demonstrates the message of system evolution.
- $s_0 \in$ State is the initial state of the process.

The behaviour of a process can be modelled by an *LTS*. Each *LTS* starts from an initial state which is a special state without any pre-state. Any states in the *LTS* can be reached from the initial state via a number of transitions. Each transition consists of three factors: s and $s' \in$ State which represents start state and end state of the transition,

respectively, and $l \in$ Label which shows the information of the state change. Hence, $(s,l,s') \in \rightarrow$ (or expressed as $s \xrightarrow{l} s'$) represents a transition in which the state of process will evolve from state s to state s', and the message involving such evolution is contained in label l.

There are basically two types of transitions in the *LTS*: those introduced by visible actions and those introduced by invisible actions. The latter are actions occurring in the process computation and indeed lead to no state change or invisible state change. Such an action is also called internal action or silent action due to the fact that it is invisible to observers. A special symbol τ is given to represent this kind of actions. Although τ actions are less significant factors in *the LTS* and finally the detail of such internal communications will be abstracted away, it is indispensable while a valid *labelled transition system* is being constituted.

Any transitions which are induced by the visible actions may be blocked by the execution environment whereas the invisible action τ will never be blocked. The mechanism to achieve such controls will be discussed in later Chapters. Therefore, once the process has made a decision to choose a particular transition, whenever it is not blocked by the underlying environment, the process will be allowed to forward to another state. *LTS* can model the process computation as sequences of transitions. Most likely, the execution of a process may contain an infinite number of transitions.

Again, for simplicity, we only consider the finite execution, which means such an execution will always reach an end after a certain number of transitions. The *LTS* which models the behaviour of the processes can be represented as a graph. In such a graph, the nodes are used to represent states of the process, and the edges are used to represent events (or transitions). These events (or transitions) usually bring out the conversion of the states, and the names of events are labelled on the edges.

3.3 Equivalence relations

Since the transitions in the derived *LTS* may contain a large number of τ transitions which are irrelevant to the desired control model, the *LTS* has to be further simplified by removing such τ transitions.

To fulfill such simplification, an appropriate equivalence relation must be chosen. In fact, a lot of equivalence notations have been applied and can be found in the literature [21, 35, 37, 53, 54]. Such equivalence relations include *trace equivalence*, *bisimilation*, and *testing equivalence*. *Bisimilation* and *testing equivalence* are somewhat strict relations while determining the equivalence, and *strong bisimilation* which is one type of *bisimilation* even takes consideration of the internal action τ as other visible actions. Due to the purpose of our testing control tool, it is not necessary for such a tool to distinguish either the program state or the set of possible next actions of the program. Therefore, *trace equivalence,* which is considered as the simplest equivalence concept, is sufficient to perform the task of simplification.

Commonly, a trace of a process is referred to a sequence of actions that such a process can execute. Thus, two states p and q in the program computation are considered to be *trace equivalent* if for all sequences of actions w, the succeeding state of state p is an accepting state if and only if the succeeding state of state q is an accepting state, where the accepting state means the final state of the program computation [23].

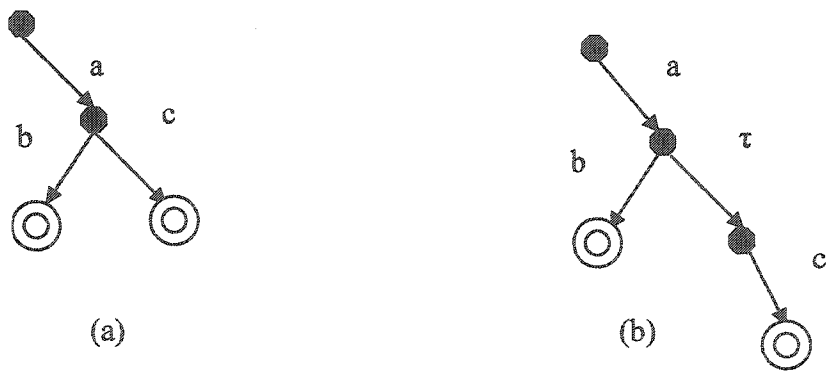

Figure 1: Illustration of Trace Equivalence

To illustrate, we can consider the graphs in **Figure 1**. It is quite obvious that there are 2 traces (or sequences of actions) {ab, ac} from the state represented by the top node to the accepting states in the **(a)** graph, whereas the traces from the top node of **(b)** graph to the accepting states are {ab, aτc}. Since the internal action τ is not observable, such

traces can be simply ignored. Hence, the traces of **Figure 1 (b)** are actually {ab, ac}, too. According to the definition, two states expressed by the top nodes in these two sub-graphs are trace equivalent since for all traces *ab* and *ac,* both of these two states can reach the accepting states.

3.4 Process terms

Beyond such important background knowledge stated above, it is essential to introduce some previous works discussed in [9]. These works, which include facilitating a specific format of process terms that will be mentioned in this section, giving a case of *LTS* and defining a number of rules for generating the *LTS* that are going to be discussed in next section, are considered as the basis of the implementation provided in this thesis work.

To model the behaviour of PUT, an efficient and effective method is necessary. One specification language, *process terms*, is such a suitable candidate to express the design specification of a software system. In fact, such *process terms* are based on the *process algebra* which is an algebraic approach to the study of concurrent processes [35].

As mentioned earlier in ***Chapter 1***, a process is conceptually equivalent to a thread in a multithreaded system. The executions inside a process are considered only to be sequential, and a set of these processes consists of the entire program. For simplicity, the implementation tool concerns only static processes. In other words, we do not consider the situation in which new processes will be produced dynamically during the execution of the program. A *process term* describes the state of the behavior of a process at one particular moment, and the combination of a set of *process terms* is applied to express the state of the behavior of the whole system.

Generally, two types of synchronization controls are concerned in the concurrent systems: to guarantee the mutual exclusion and to realize the process coordination. The monitor is assumed as the key mechanism to handle the synchronization activities in this thesis, and each of which provides following functionality:

- Each monitor maintains a lock and a queue for this lock. The lock is used to ensure the exclusive access to the critical sections, and the lock queue is

employed to contain a sequence of processes that intend to access the critical section protected by such a monitor.

- Two operations *wait* and *notify* can be performed on the monitors in order to accomplish the coordination and cooperation among various processes. Monitors may have condition variables, on which a process can execute *wait* to give up the lock and put itself into the waiting queue of this monitor if conditions are not right for it to continue executing. A process will not be able to continue its execution when it waits in the monitor-waiting queue. Later on, another process may execute *notify* to wake up and remove the first process in the waiting queue of a monitor if such a queue is not empty. The awakened process will be re-enabled for its execution and compete with others for the lock of this monitor. One additional operation *notifyAll,* which is a special case of *notify* operation and wakes up all the waiting processes in the queue instead of only the first one, will also be considered here.

By assumption, it is given that a set V of variables, a set MID of monitor identifiers where MID $\subset$ N that indicates all the monitor identifiers must be non-negative integers, and the same is true for a set PID of process identifiers where PID $\subset$ N.

The following BNF (or Backus Naur Form, which is a formal notation to describe the syntax of a given language) gives the structure of a process term p:

p = stop | s; p

s = x := e | if c then q_1 else q_2 | while c do q | input(x) | lock(m, q) |
wait (m_1, m_2) | notify (m) | notifyAll(m)

q = s | s;q

where $x \in$ V indicates that x is one of the variables in the set V; m, m_1, $m_2 \in$ MID which indicates that m, m_1 and m_2 are monitor identifiers; s represents a statement which indeed can be considered as a type of design abstract rather than the actual program code; q, q_1, and q_2 are intermediate sequence of statements; and c is a *Boolean* expression over V.

Intuitively, the first equation claims that a process term consists of either a *stop,* which is a special statement that indicates the action of ending the process execution, or a

statement *s* followed by the rest of this process term p. In other words, a process term is built up from a set of statements including *stop*.

The second equation in the above BNF defines that a statement *s* may be one of the following eight formats:

- *x*:=*e*: is an arithmetic assignment operation, where e is an arithmetic expression over V which indicates that each variable must have already been defined in V if it appears in the *e*.
- *if c then* q_1 *else* q_2: is a two-armed conditional expression, which means as long as the condition *c* is satisfied, the statement segment q_1 will be executed; otherwise, the statement segment q_2 will be executed. In particular, a one-armed conditional expression, in which the segment q_1 or q_2 may be empty, is also a legal format of the statement.
- *While c do q*: is a repetition structure, which means as long as the condition *c* is satisfied, the statement segment *q* will be executed repeatedly.
- *Input(x)*: means to get the value from the sequence of input which will also be given, and assign this value to the variable *x*.
- *Lock (m, q)*: means that the monitor *m* is applied to ensure the mutual exclusion of the execution of statement segment *q* of the process.
- *Wait* (m_1, m_2): means the action of releasing the occupied lock on monitor m_1 and putting the current running process on the waiting queue of monitor m_2; later on it will re-acquire the lock of monitor m_1 after it is woken up by another process.
- *Notify (m)*: means the action of waking up and removing the first process on the waiting queue of monitor *m*.
- *NotifyAll(m)*: means the action of waking up and removing all the processes on the waiting queue of monitor *m*.

The third equation in the above BNF indicates the structure of segment *q*. The segment *q* can either be a statement or a statement followed by the rest of segment *q*. In other words, the segment *q* is made up of a sequence of statements.

3.5 Structural operational semantics and the structural rules

As mentioned earlier in ***Chapter 1,*** the present work considers that the structural operational semantics is used to systematically and automatically produce the *Labelled Transition System* which includes necessary information to retrieve all feasible *path constraints*, and a brief introduction of *LTS* is given in ***Section 3.2***. From the design abstracts, which are constructed by *process terms* introduced in last section, a specific case of *LTS* has to be applied, and certain rules are indispensable for generating such a *LTS* [9].

3.5.1 A case application of LTS

It has already been defined in ***Section 3.4*** that *V* is a set of variables, *MID* is a set of monitor identifiers, and *PID* is a set of process identifiers. Now, let *I* represent the set of sequences of input values, *I'* represent a combination of the set of input values and the set of variable values, and *P* represent the set of process terms over *MID* and *V*. This means such process terms will alter the state of PUT according to the status of certain monitors and variables. $E \subseteq V \rightarrow I'$ indicates the mapping from variables defined in set V to their values contained in set I'; $L \subseteq MID \rightarrow \{true, false\}$ indicates the set of monitors along with their lock status, where value *true* denotes that the lock of this monitor is occupied by a process, and value *false* denotes that the lock of this monitor is currently available; $Q \subseteq MID \rightarrow PID^*$ indicates the mapping from the monitors to a sequence of processes that are currently waiting in the waiting queues of such monitors, where PID* represents a sequence of 0 or more process identifiers; and $PR \subseteq 2^{PID \times P}$ indicates the set of states of the process which possibly contains $2^{PID \times P}$ elements.

Thus, the definition of *LTS* specified in ***Section 3.2*** can be applied as the following case:

A *labelled transition system* (*LTS*) is a quadruple <State, Label,→, s0>, where

- State $\subseteq I \times E \times L \times Q \times PR$, which means the state has to contain the information including a sequence of input values, the mapping from variables to their values, the locking status of all monitors, the status of monitor waiting queue, and the status of processes, from set I, E, L, Q, and PR, respectively;
- Label $\subseteq (PID \times \{lock, wait, notify\} \times MID) \cup (PID \times \{input\}) \cup \{\tau\}$, which indicates only two types of events: synchronization events and input events, which are considered to be the content of a label as mentioned earlier, and internal events τ in which is only used for the computation and will be removed afterward;
- $\rightarrow \subseteq State \times Label \times State$ is a set of transitions with its actions described by labels.
- $s0 \in State$ is the initial state of the processes.

By assumption, the *test case* will be given as mentioned in ***Chapter 1***. Such a *test case* is made up of two parts: an input sequence and an expected output. Thus, the input sequence contained in the system states of *LTS* can be derived from the given *test case* by taking away the part of output. In fact, the input is one of the key points for constructing *LTS* and the control models since the latter ones will most likely differ from each other due to different inputs. Since the program may consist of more than one process, the status of all such processes must be included in the state of *LTS*, and each process has to be marked by its process identifier which uniquely distinguishes it from other processes in the program. In other words, every state which describes the behaviour of whole program system is made of different pieces of information that describe the behaviour of every process and other type of information.

3.5.2 Structural rules

Based on the particular case described in ***Section 3.5.1***, the schema of structural rules and the details of such rules will be introduced as follows.

3.5.2.1 Schema for structural rules

To construct the *LTS*, the transition relation → is another key. There are always two states involved in such a transition: a start state and a next state. For the implementation in this thesis, a set of structural rules is defined for the transition relation →. The schema for such structural rules is $\frac{ANTECEDENT}{CONSEQUENT}$, which is logically equivalent to $\forall$ (ANTECEDENT $\rightarrow$ CONSEQUENT) and can be paraphrased to indicate that for all the relation of ANTECEDENT implying CONSEQUENT. Also, the ANTECEDENT and CONSEQUENT share free variables; thus they will be treated as *true* in case the ANTECEDENT is absent. On the other hand, the semantics considered by this *LTS* are interleaving, which means only one process among all the processes is allowed to perform one of its statements at a given time. The reason for following such interleaving semantics is not only that the semantics are very simple but also that the sequential control is actually the only control mechanism which can be accomplished by the current control tool. In fact, the control tool is not able to fire two events at the same time. Consequently, one important feature is shared by all these structural rules: the evolution of the system state only concerns the operation of one of the processes at one step whereas the others keep still.

According to their functionalities, the structural rules can be grouped into four sets, and are illustrated below.

3.5.2.2 Structural rules for basic flow of controls

The first set of structural rules concerns the common flow of control and consists of the following 5 rules:

Assignment Rule:

$$\frac{(x, f) \in E}{\langle I, E, L, Q, P \,\|\, pid : (x := e); p\rangle \xrightarrow{\tau} \langle I, E[x / Eval(E, e)], L, Q, P \,\|\, pid : p\rangle}$$

Condition True Rule:

$$\frac{Eval(E,c) = true}{\langle I,E,L,Q,P \parallel pid : (if\,(c)then(p1)else(p2)); p3\rangle \xrightarrow{\tau} \langle I,E,L,Q,P \parallel pid : p1; p3\rangle}$$

Condition False Rule:

$$\frac{Eval(E,c) = false}{\langle I,E,L,Q,P \parallel pid : (if\,(c)then(p1)else(p2)); p3 > \xrightarrow{\tau} \langle I,E,L,Q,P \parallel pid : p2; p3\rangle}$$

Loop Continue Rule:

$$\frac{Eval(E,c) = true}{\langle I,E,L,Q,P \parallel pid : (while(c)do(p1)); p2 > \xrightarrow{\tau} \langle I,E,L,Q,P \parallel pid : p1; (while(c)do(p1)); p2\rangle}$$

Loop End Rule:

$$\frac{Eval(E,c) = false}{\langle I,E,L,Q,P \parallel pid : (while(c)do(p1)); p2 > \xrightarrow{\tau} \langle I,E,L,Q,P \parallel pid : p2\rangle}$$

Figure 2: Structural rules for basic flow controls

Since there will probably be a number of processes in the program, pid_1:$p1$ || pid_2:$p2$ || --- || pid_n:pn is used to denote each process which has a process identifier (pid_1, or pid_2, ---, or pid_n) and is described by a process term ($p1$, or $p2$, ---, or pn), and the symbol "||" indicates that the processes separated by it exist simultaneously. In the notation of these rules, P || pid: a; p represents a set of processes including process term a:p which has the process identifier pid and some other processes expressed in P. Specifically, the process term a:p denotes a process term p which follows a statement a, where the *statement* is actually some kind of a design abstract rather than a program code and represents an assignment, a choice, or a while-loop. The definitions of x, e, and c are given in ***Section 3.4***, and *Eval(E,e)* and *Eval(E,c)* are used to represent the evaluation of e and c based on the variable to value mapping set E, respectively. *E[x/v]* indicates that the value of the variable x is replaced by v in the variable to value mapping set E.

The first rule involves the system evolutions introduced by an assignment statement. According to the definition of *LTS*, each state in *LTS* consists of the factors *I*,

E, L, Q, and *P,* which have been described earlier. To apply the first rule, the start state has to meet certain criteria: a variable *x* and its value *f* are described in E, and the current statement is an assignment *x:=e* followed by a *process term p* in the process that is identified by *pid.* Thus, by performing this assignment statement, the system can be moved to the next state. All the factors of this next state are exactly same as those of the start state except for the following changes: the value of variable *x* is replaced by *v* in E, and the current statement to be executed in the process *pid* becomes the first statement of the *process term p* (or say *process term p* instead). The second and third rules concern the conditional statements. To apply the second rule, the start state has to meet certain criteria: a *Boolean* expression *c* over *V* is evaluated to be *true* based on *E,* and the current statement to be executed is a conditional statement, which is determined by condition *c* and has two succeeding branches: *process terms p1* and *p2,* and followed by another *process term* p3. Thus, by performing the conditional statement, the system can be moved to the next state. All the factors of this next state are exactly same as those of the start state except that the current statement of the process *pid* becomes *p1* followed by *p3.* In contrast, if such a *Boolean* expression *c* over *V* is evaluated to be *false* based on *E,* the third rule can be used. By performing the conditional statement, the system can be moved to the next state in which the current statement of the process *pid* is *p2* followed by *p3.* The fourth and fifth rules concern the *while-loop* statements. To use the fourth rule, the start state must satisfy the prerequisite: a *Boolean* expression *c* over *V* is evaluated to be *true* based on *E,* and the current statement to be executed is a *while-loop* statement, which contains a *process term p1* and will be ended whenever the condition *c* becomes *false,* followed by another *process term p2.* Thus, by performing the *while-loop* statement, the system can be moved to the next state. All the factors of this next state are exactly same as those of the start state except that the current statement of process *pid* is the *process term p1* followed by such a while-loop statement that contains *p1* and is determined by *c,* and in turn this while-loop statement is followed by *p2.* On the other hand, if such a *Boolean* expression *c* is evaluated to be *false,* the fifth rule can be applied. By performing the *while-loop* statement, the system can be moved to the next state in which the current statement of process pid is the *process term p2.*

In essence, this set of structural rules causes only the system evolutions which are not involved in producing the *path constraints*. In other words, these evolutions are invisible so each transition is labelled by a τ.

3.5.2.3 Structural rules for input action

As mentioned earlier, the input action is one of the interesting points in the study of *path constraints* generation. The rule defined below involves the action of input.

Input Receive Rule:

$$\frac{I \neq \phi}{\langle I, E, L, Q, P \| \, pid : input(x); p \rangle \xrightarrow{(pid, input)} \langle rest(I), E[x / first(I)], L, Q, P \| \, pid : p \rangle}$$

Figure 3: Structural rule for input

The prerequisite for applying this input receive rule is that the input sequence *I* cannot be empty. Thus, if the current statement to be executed is an input statement that reads a value into variable *x,* and this statement is followed by a *process term p,* the system state can be moved to the next state by performing the input action. All the factors of this next state are exactly same as those of the start state except that the value of variable *x* in E is changed to the first data in I while the first data has been removed from I, the current statement of the process *pid* becomes *p*. This transition is labelled by *(pid, input)* where the *pid* is the identifier of the process which contains such an input statement.

3.5.2.4 Structural rules for mutual exclusion

Apart from the action of input, synchronization activities of the PUT are important places to be explored in generating *path constraints*, too. Such synchronization activities can be grouped into two aspects: those for mutual exclusion and those for process coordination. The set of structural rules involved with mutual exclusion will be discussed

in this section, and the rules about process coordination will be addressed in the next section.

In order to describe the activities of the monitors and support the simulation of the computational behaviour of PUT, it is necessary to introduce the following additional internal statements:

- *lock_restart(m)*: means the action of a process to re-acquire the lock on the monitor *m* after this process is being notified by another process from its waiting status;
- *lock_end(m)*: means the process completes a critical section which is controlled by monitor *m*;
- *waiting(m)*: means the process remains in the waiting status on the waiting queue of monitor *m*.

Hence, to simulate the mutual exclusion in a process execution, the following set of rules is given.

Lock Begin Rule:

$$\frac{Eval(L,m) = false}{\langle I,E,L,Q,P \parallel pid : lock(m,p1); p2\rangle \xrightarrow{(pid,lock,m)} \langle I,E,L[m/true],Q,P \parallel pid : p1; lock\_end(m); p2\rangle}$$

Lock Restart Rule:

$$\frac{Eval(L,m) = false}{\langle I,E,L,Q,P \parallel pid : lock\_restart(m); p\rangle \xrightarrow{\tau} \langle I,E,L[m/true],Q,P \parallel pid : p\rangle}$$

Lock End Rule:

$$\frac{Eval(L,m) = true}{\langle I,E,L,Q,P \parallel pid : lock\_end(m); p\rangle \xrightarrow{\tau} \langle I,E,L[m/false],Q,P \parallel pid : p\rangle}$$

Figure 4: Structural rules for mutual exclusion

In the notation of these rules, *Eval(L,m)* denotes the evaluation of lock status on monitor m in L, where *true* indicates that the lock is currently occupied by another

process and *false* indicates that the lock is available; *L[m/v]* denotes that the value of the lock status on the monitor *m* has been changed to *v* in L.

To apply the first rule in this set, the start state has to meet the criteria: the lock of m is available, the current statement to be executed is a *lock* statement that tries to execute a *process term p1* in a critical section which is ensured by the lock of *m,* and this *lock* statement is followed by another *process term p2.* Thus, this start state can be moved to next state by performing the *lock* statement. All the factors of this next state are exactly same as those of the start state except that the lock status of *m* has been changed to *true* in L and the current statement of the process *pid* becomes *p1* followed by a *lock-end* statement which is inserted manually and followed by *p2,* in turn. The transitions following the Lock Begin Rule will be labelled by *(pid, lock, m)* where *pid* is the identifier of the process that contains the *lock* statement and *m* is the identifier of the monitor that contains this lock. By artificially inserting this *lock_end* statement, the releasing of the lock becomes observable, which is necessary for the generating *of LTS*; otherwise, there is no way to detect such an activity. The Lock End Rule is defined to express the transitions moved by such a *lock_end* statement. In particular, if the lock of *m* is currently occupied, and the current statement to be executed is a *lock_end* statement that is going to release the lock on monitor *m* followed by another *process term p,* the system state can be moved by performing the *lock_end* statement to next state. All the factors of this next state are exactly same as those of the start state except that the lock status of *m* has been changed to *false* in L, and the current statement of the process *pid* becomes *p*. Since the *lock_end* statement is only used for the computation of generating the *LTS*, and such information has nothing to do with the *path constraints*, the transitions caused by the *lock_end* statement will be labelled by a τ.

The Lock Restart Rule is defined to express the transitions moved by another internal statement: the *lock_restart* statement. This rule indicates if the lock of *m* is available, the system state, in which the current statement to be executed is a *lock_restart* statement that tries to regain the lock of *m* and is followed by the process term *p,* can be moved by performing the *lock_restart* statement to the next state in which the lock status of *m* has been changed to *true* in L, and the current statement of the process *pid* is *p*.

Similarly, since the *lock_restart* information helps to build only the *LTS*, the transition caused by *lock_restart* statement will be labelled by a τ.

3.5.2.5 Structural rules for process coordination

The last but not least important set of structural rules deals with the coordination between different process, and is defined as follows.

Wait Rule:

$$\langle I, E, L, Q, P \parallel pid : wait(m_1, m_2); p\rangle \xrightarrow{(pid, wait, m2)} \langle I, E, L[m_1 / false], enqueue(Q, m_2, pid), P \parallel pid : p'\rangle$$

where p'=waiting (m_2); lock_restart(m1); p.

Notify With Nonempty Queue Rule:

$$\frac{first(Q, m) = pid_2}{\langle I, E, L, Q, P \parallel pid_1 : notify(m); p1 \parallel pid_2 : waiting(m); p2\rangle \xrightarrow{(pid, notify, m)} \langle I, E, L, dequeue(Q, m), P \parallel pid_1 : p_1 \parallel pid_2 : p2\rangle}$$

Notify with Empty Queue Rule:

$$\frac{first(Q, m) = null}{\langle I, E, L, Q, P \parallel pid_1 : notify(m); p\rangle \xrightarrow{(pid, notify, m)} \langle I, E, L, Q, P \parallel pid_1 : p_1\rangle}$$

Figure 5: Structural rule for processes coordination (a)

In these rules listed in **Figure 5**, *enqueue(Q, m, pid)* is used to express the set of waiting queues derived from the set Q which is the mapping from monitors to their waiting queues. These waiting queues contain the sequences of processes currently waiting for the locks on these monitors after adding *pid* into the waiting queue of monitor *m*. Similarly, *dequeue(Q,m)* is used to express the set of waiting queues from Q by removing the first element from the waiting queue of *m*, and *first(Q, m)* is used to represent the first process on the waiting queue of *m* in Q.

To use the first rule in this set, the start state must satisfy the prerequisite: the current statement to be executed is a *wait* statement that is going to release the lock on monitor m_1 and put the current running process into the waiting queue of monitor m_2, and

this *wait* statement is followed by a *process term p*. Thus, by performing this *wait* statement, the system state can be moved to next state in which the lock status on m_1 has been changed to *false* in L, the set Q has been modified by adding *pid* into the waiting queue of m_2, and the current statement of the process *pid* becomes the statement *waiting*(m_2) followed by the statement *lock_restart*(m_1) that is in turn followed by *p*. In particular, the *waiting* statement and the *lock_restart* statement are artificially inserted for the purpose of constructing the LTS, where the definition of *waiting* statement and *lock_restart* statement can be found in ***Section 3.5.2.4***. The statement *Lock_restart* is used after the *waiting* statement in order to enable the waiting process to regain the lock after it is notified by another process. The transitions evolved according to this rule are labelled by *(pid, wait, m_2)*.

The second rule is a special one which involves the actions of two different processes. To apply this rule, the start state has to meet certain criteria: the waiting queue of the monitor *m* is not empty and the first process in this waiting queue in Q is the process pid_2; the current statement to be executed is a *notify* statement in the process pid_1 that is going to inform the first process in the waiting queue of the monitor *m;* and there exists another process pid_2 that is currently waiting as the first element in the waiting queue of the monitor *m* and has a succeeding process term p_2. Thus, the system state can be moved by performing the *notify* statement to next state in which the first element on the waiting queue of m has been removed, and the current statement of process pid_1 and pid_2 become *p1* and *p2*, respectively. The transitions moved by this rule are labelled by *(pid, notify, m)*. This rule enables two processes to move to their next states simultaneously, which simulates the typical hand-shaking mechanism between synchronization processes in concurrent programs. However, since it is not necessary to pay more attention to control the statement *waiting(m)* due to the fact that it is not the event related to the *path constraints* generating, the notifying process can be considered as the only factor to cause the system evolution. Therefore, with such acknowledgement, there will be no contradiction between this *notify* rule and the test control mechanism expressed earlier which assures that the transitions in the LTS will not be stimulated by more than one event at one time.

The third rule actually deals with an extreme case when performing the *notifying* statement. The prerequisite for applying this rule is that the waiting queue of monitor *m* is empty in Q, the current statement to be executed is a *notify* statement that is going to inform the first process in the waiting queue of *m* and is followed by another *process term p*. Thus, the system state can be moved by performing the *notify* statement to next state in which the current statement of the process pid_1 becomes *p*. Indeed, such move has nothing to do with the change of system state. The transitions stimulated by this rule are labelled by *(pid, notify, m)*, too.

3.5.2.6 Extension of structural rules for process coordination

Although two *notify* rules have been defined in [55], there is no such a structural rule to consider the more specific case of *notify*--- *notifyAll*. To generate *LTS* for the PUT which contains the action of notifying all the processes in the waiting queue of a monitor, additional structural rules are needed to be extended.

NotifyAll with nonempty Queue Rule:

$$\frac{EvalEmpty(Q,m) = false}{\langle I,E,L,Q,P \parallel pid_1 : notify(m); p1 \parallel pid_2 : waiting(m); p2 \parallel \cdots \parallel pidn : waiting(m); p_n\rangle \xrightarrow{(pid,notifyAll,m)} \langle I,E,L,dequeueAll(Q,m),P \parallel pid_1 : p_1 \parallel pid_2 : p2 \parallel \cdots \parallel\!\parallel pidn : p_n\rangle}$$

NotifyAll with Empty Queue Rule:

$$\frac{EvalEmpty(Q,m) = true}{\langle I,E,L,Q,P \parallel pid_1 : notifyAll(m); p\rangle \xrightarrow{(pid,notifyAll,m)} \langle I,E,L,Q,P \parallel pid_1 : p_1\rangle}$$

Figure 6: Structural rule for processes coordination (b)

In the notation of these two rules in **Figure 6**, *EvalEmpty(Q,m)* is defined to express the evaluation of the waiting queue status of *m* in Q where the value *true* indicates the queue is empty; and *dequeueAll(Q,m)* is defined to express the set of waiting queues in Q after removing all elements from the waiting queue of *m*.

To apply the first NotifyAll rule, the start state has to meet certain criteria: the waiting queue of the monitor *m* in Q is not empty; the current statement to be executed is a *notifyAll* statement of the process pid_1 that is going to inform all processes in the waiting queue of *m*; *and* there exist processes pid_2, ---, pid_n that are currently waiting on monitor *m* and have succeeding *process terms* p_2, ---, *pn*. Thus, the system state can be moved by performing the *notifyAll* statement to the next state in which all the elements on the waiting queue of *m* have been removed, and the current statements of process pid_1, pid_2, ---, pid_n become *p1*, *p2*, ---, *pn*, respectively. The transitions moved by this rule will be labelled by *(pid, notifyAll, m)*.

The second rule is analogous to the rule of *notify* with empty queue. Similarly, the prerequisite to use this NotifyAll rule is that the waiting queue of monitor *m* in Q is empty, and the current statement to be executed is a *notifyAll* statement followed by another *process term p*. Thus, the system state can be moved by performing *notifyAll* statement to next state in which the current statement of process pid_1 is *p*. The transitions moved by this rule will be labelled by *(pid, notify, m)*.

So far, all the structural semantics rules have been introduced, which are sufficient to derive the LTS from the given design abstract. To demonstrate the procedure of LTS generating, an example will be given in ***Chapter 4***.

Chapter 4 An Example

To better illustrate the domain of our specific problem on testing, one typical concurrent example called *producer consumer problem* is provided in this Chapter. In such an example and others, it is assumed that the design abstract of the program in our test environment is given in terms of process terms which are presented in ***Chapter 3.4***.

4.1 The problem of producer consumer

The *producer consumer problem* is a classic problem that concerns synchronization. There are two types of processes in such a problem: producer and consumer. The producer and consumer processes share a common bounded buffer. The producer executes an infinite loop where it puts new items into the buffer, whereas the consumer exercises an infinite loop where it removes items from the buffer.

To give the solution to the *producer consumer problem*, two important aspects have to be considered:

(1) mutual exclusion: at most one process that is either producer or consumer can access the shared buffer at one time;

(2) synchronization: the producer and consumer processes have to check the content of the buffer before performing the action of depositing and withdrawing; in particular, the producer can deposit only if not all the slots of the buffer are full, and consumer can withdraw only if not all the slots of the buffer are empty. Otherwise, the producer or consumer has to put itself into a waiting status until the condition is satisfied.

The concurrent programs normally consist of a number of process types. For example, there are two types of processes which are *producer* and *consumer* in the *producer consumer problem*. However, each process type may have more than one instance. For example, there may exist two *producers* and three *consumers* in the *producer consumer problem*. In fact, it is important to determine the number of these instances. On one hand, it is difficult to handle large number of instances and perform the

thorough test of a concurrent program. On the other hand, these numbers cannot be too small to avoid all possible faults to be disclosed in the testing. Also, it is obviously not appropriate to allow the producer or consumer to loop infinitely for the purpose of testing since it is mentioned earlier in this thesis that our testing will only deal with programs that will terminate. Hence, for the sake of simplicity and without losing the generality, the *producer consumer problem* illustrated in this thesis will consider one instance of producer and two instances of consumer. Again, for simplicity reasons, it is assumed that the shared buffer can only contain two items, and the loop of the producer is limited up to 3 times. Thus, upon execution, the producer will take three inputs of integer numbers and deposit them to a two-slot bounded buffer, and two consumers will withdraw these integer numbers from the bounded buffer.

4.2 Design abstract for one solution of producer consumer problem

The *design abstract* for our solution of *producer consumer problem* in this thesis is displayed in **Figure 7**. This *design abstract* code is given in terms of process terms.

```
* Producer&Consumer *
<variables> {{ x :int },{ buffer0:int },{ buffer1:int },{ count :int },{ times :int }}
<monitors> {m0,m1,m2}

<Process type>: Producer
while ( times < 3 ) do {
    input ( x );
    lock ( m0 ) {
        while ( count == 2 ) do {
            wait ( m0 , m1 );
        }
        if ( buffer0 == 0 ) then {
            buffer0 := x;
        } else {
            buffer1 := x;
        }
        count := count + 1;
        times := times + 1;
        notifyAll ( m2 );
    }
    lock_end ( m0 );
```

```
}
stop;

<Process type>: Consumer
while ( true ) do {
    lock ( m0 ) {
        while ( count == 0 ) do {
            if ( times == 3 ) then {
                lock_end ( m0 );
                stop;
            }
            wait ( m0 , m2 );
        }
        if ( buffer0 != 0 ) then {
            buffer0 := 0;
        } else {
            buffer1 := 0;
        }
        count := count - 1;
        notify ( m1 );
    }
    lock_end ( m0 );
}

<process> 1: Producer
<process> 2: Consumer
<process> 3: Consumer
```

Figure 7: Design abstract for *Producer Consumer problem*

First of all, each design abstract will be given a name which is placed in the first line and specified between two "*", in this case, *Producer&Consumers*. The second thing in the design abstract is to provide the declarations of all the variables and monitors. The declaration of a variable consists of the variable name and its type, where the variable types considered in the scope of this thesis work are only *integer* numbers denoted by *int*, *boolean* which has value of *true* or *false*, and *string* which is a sequence of characters.

In general, since the program consists of processes, the design abstract may also have a number of modules and each one is used to describe the functionality of a specific process type. Such a module starts with a signature *<Process type>* followed by a

process type name. For simplicity, it is considered that the set of processes of the *PUT* will be given statically in the design abstract. Even though there are three total processes (one *producer* and two *consumers*) in the current version of *producer consumer problem*, two process types - *producer* and *consumer* - will be described in the design abstract due to the fact that two consumer processes are indeed exactly same, except for the process identifiers. After giving the process types, the description of each process is provided in terms of these types.

The solution to the *producer consumer problem* described by this design abstract employs monitor *m0* for guaranteeing the mutual exclusion and monitors *m1*and *m2* for achieving the process coordination. Specifically, *m0* ensures the exclusive access to the shared buffer and modification to the buffer count; *producer* checks the buffer count before producing data into the buffer, and releases the acquired lock on monitor *m0* and waits in the waiting queue of monitor *m1* if all slots of the shared buffer are full; *consumer* checks the buffer count before consuming data from the buffer, and releases the acquired lock on monitor *m0* and waits in the waiting queue of monitor *m2* if all slots of the shared buffer are empty. Meanwhile, *producer* takes the responsibility to wake up all the *consumer*s that are waiting in the queue of monitor *m2* after it has produced data into the buffer; *consumer* is responsible to wake up the *producer* that is waiting in the queue of monitor *m1* after it has consumed data from the buffer. On the other hand, the variables used in this design abstract are: *x* that is used to contain the input value; *buffer0* and *buffer1* that are used to contain the value of the buffer; *count* that is used to count the number of full slot of the buffer; and *times* that is used to count the times of producing the new data by the producer. The type of these variables is integer, and the initial value of each is 0.

4.3 Generation of LTS

According to the structural rules provided in ***Chapter 3.5*** and a given initial state, the LTS can be constructed from the design abstract due to the fact that all the states in the LTS are reachable from the initial state via the transitions conducted by such rules.

To illustrate the process of generating LTS, the design abstract given in ***Section 4.2*** will be used as an example. Assume that the initial state is *S0*: < I, E, L, Q, P >, where

- I = <1, 2, 3> that denotes there are totally three inputs: *1, 2,* and *3*;
- E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, that denotes the variables used in this program are *x, buffer0, buffer1, count,* and *times*, and their initial values are *0, 0, 0, 0, 0,* respectively;
- L = {(m0, false), {(m1, false), {(m2, false)} that denotes the monitor used in this program is *m0, m1, and m2*, and their locks are all available initially;
- Q = {(m0,<>), (m1, <>), (m2,<>)} that denotes there exist three monitors *m0, m1, and m2,* and their waiting queues initially all contain no element;
- P consists of process terms for all the processes, in particular, *process 1*, *process 2* and *process 3*.

As mentioned earlier in ***Section 4.2***, symbol "||" is used to denote that all such processes exist and execute at the same time. Also, assume *p1*, *p1'*, *p2*, *p2'* are succeeding process terms after the current statement of process type *producer* and *consumer*, respectively. Thus, the partial results of deriving the *LTS* for the *producer consumer program* are as shown below.

S0: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***while (times <3) do p1*** || 2 : while (true) do p2 || 3 : while (true) do p2>

↓ **τ (by performing *while_loop statement*)**

S1: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***input(x) ; p1***|| 2 : while (true) do p2 || 3 : while (true) do p2>

↓ **(1, input) (by performing *input statement*)**

S2: < I = <***2, 3***>, E = {(***x = 1***), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***lock(m0) ; p1***|| 2 : while (true) do p2 || 3 : while (true) do p2>

↓ **(1, lock, m) (by performing *lock statement*)**

S3: < I = <2, 3>, E = {(x = 1), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(***m0, true***), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***while (count == 2) do p1***|| 2 : while (true) do p2 || 3 : while (true) do p2>

↓ τ **(by performing *while_loop statement*)**

S4: < I = <2, 3>, E = {(x = 1), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, true), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***if (buffer0 == 0) then p1 else p1'***|| 2 : while (true) do p2 || 3 : while (true) do p2>

↓ τ **(by performing *conditional statement*)**

S5: < I = <2, 3>, E = {(x = 1), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, true), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***buffer0 := x; p1*** || 2 : while (true) do p2 || 3 : while (true) do p2>

↓ τ **(by performing *assignment statement*)**

S6: < I = <2, 3>, E = {(x = 1), (***buffer0 = 1***), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, true), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***count := count + 1; p1*** || 2 : while (true) do p2 || 3 : while (true) do p2>

↓ τ **(by performing *assignment statement*)**

S7: < I = <2, 3>, E = {(x = 1), (buffer0 = 1), (buffer1 = 0), (***count = 1***), (times = 0) }, L = {(m0, true), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***times := times + 1; p1*** || 2 : while (true) do p2 || 3 : while (true) do p2>

↓ τ **(by performing *assignment statement*)**

S8: < I = <2, 3>, E = {(x = 1), (buffer0 = 1), (buffer1 = 0), (count = 1), (***times = 1***) }, L = {(m0, true), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***notifyAll(m2); p1*** || 2 : while (true) do p2 || 3 : while (true) do p2>

↓ **(1, notifyAll, m) (by performing *notifyAll statement*)**

S9: < I = <2, 3>, E = {(x = 1), (buffer0 = 1), (buffer1 = 0), (count = 1), (times = 1) }, L = {(m0, true), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***lock_end(m0); p1*** || 2 : while (true) do p2 || 3 : while (true) do p2>

↓ τ **(by performing *lock_end statement*)**

S10: < I = <2, 3>, E = {(x = 1), (buffer0 = 1), (buffer1 = 0), (count = 1), (times = 1) }, L = {(***m0, false***), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1: ***while (times <3) do p1*** || 2 : while (true) do p2 || 3 : while (true) do p2

↓ τ **(by performing *while_loop statement*)**

S11: < I = <2, 3>, E = {(x = 1), (buffer0 = 1), (buffer1 = 0), (count = 1), (times = 1) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***input(x) ; p1*** || 2 : while (true) do p2 || 3 : while (true) do p2

↓ **(1, input) (by performing *input statement*)**

S12: < I = < *3*>, E = {(*x = 2*), (buffer0 = 1), (buffer1 = 0), (count = 1), (times = 1) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***lock(m0) ; p1***|| 2 : while (true) do p2 || 3 : while (true) do p2>

↓ **(1, lock, m) (by performing *lock statement*)**

⋮

⋮

Figure 8: Trace of generating LTS by depth-first traversal strategy

The procedure in **Figure 8** illustrates the generation of *LTS* by applying *depth-first* traversal strategy. Actually, such *LTS* constructed by this coarse method will be a *tree* in which the root is the initial state *S0*. This tree converted from **Figure 8** will look like the shape in **Figure 9**.

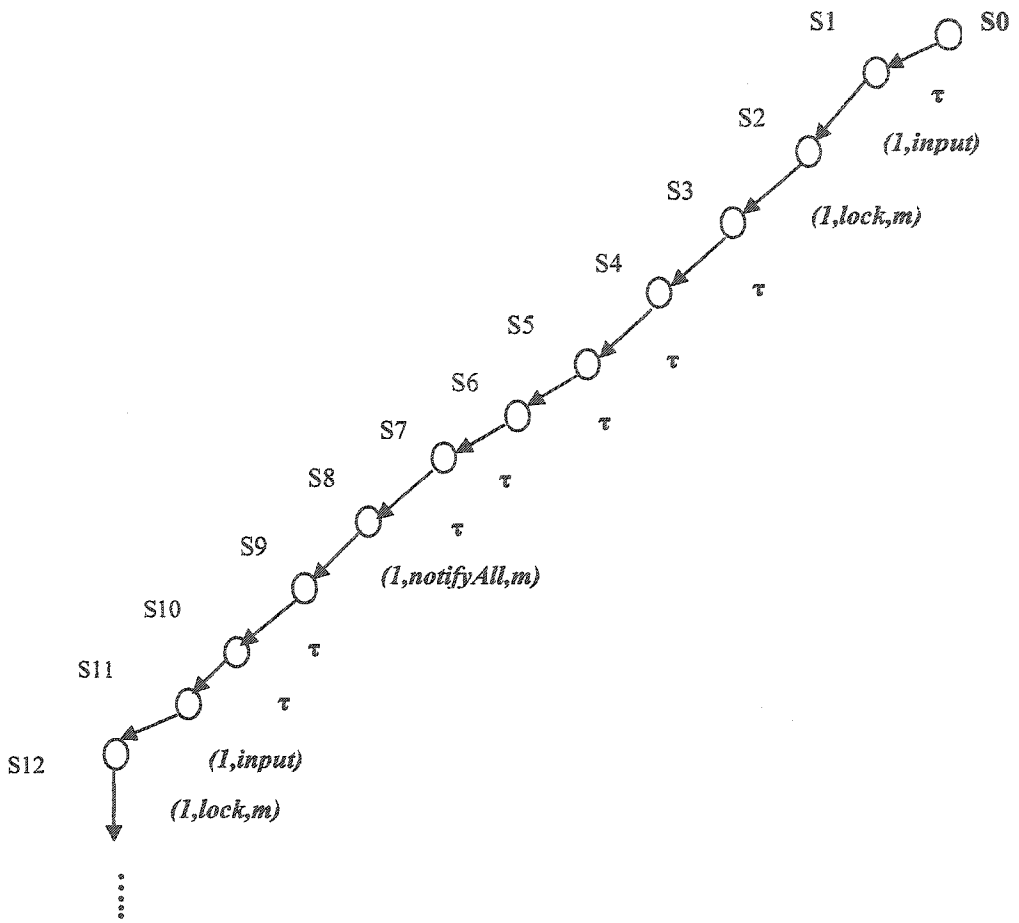

Figure 9: Tree of *LTS* generated by *depth-first* traversal strategy

Similarly, the tree which illustrates LTS generating procedure by using *width-first* traversal strategy is shown in **Figure 10**.

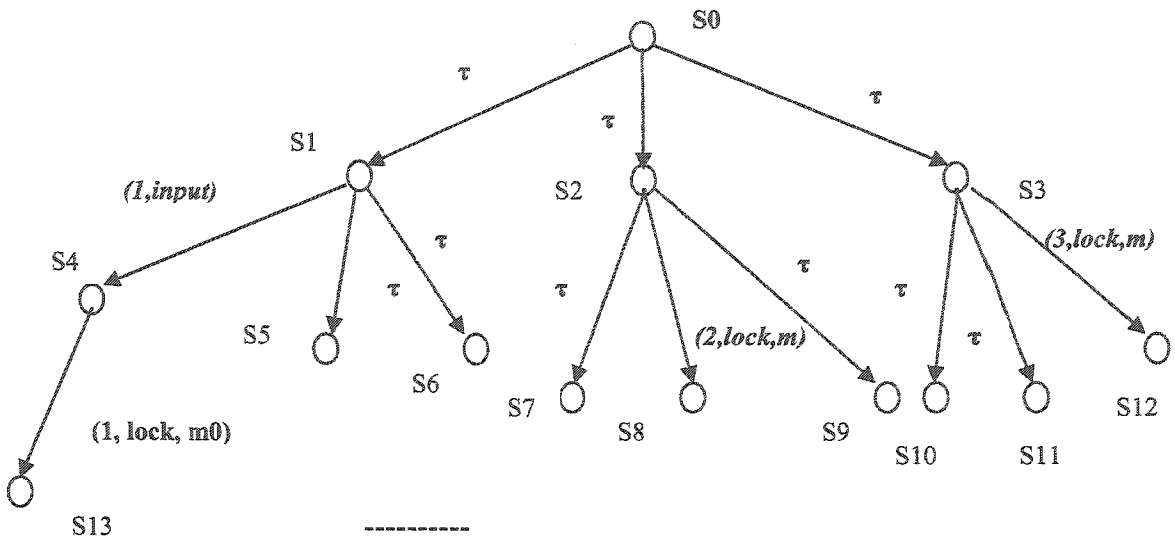

Figure 10: Tree of LTS generated by width-first traversal strategy

In **Figure 10,** the descriptions of each state are given as follows:

S0: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : while (times <3) do p1 || 2 : while (true) do p2 || 3 : while (true) do p2>;

S1: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***input(x) ; p1***|| 2 : while (true) do p2 || 3 : while (true) do p2>;

S2: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : while (times <3) do p1 || 2 : ***lock(m0); p2*** || 3 : while (true) do p2>;

S3: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : while (times <3) do p1 || 2 : while (true) do p2|| 3 : ***lock(m0); p2*** >;

S4: < I = < ***2, 3***>, E = {(***x = 1***), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***lock(m0); p1***|| 2 : while (true) do p2 || 3 : while (true) do p2>;

S5: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : input(x) ; p1|| 2 : ***lock(m0); p2*** || 3 : while (true) do p2>;

S6: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : input(x) ; p1|| 2 : while (true) do p2 || 3 : ***lock(m0); p2*** >;

S7: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***input(x) ; p1*** || 2 : lock(m0); p2 || 3 : while (true) do p2>;

S8: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(***m0, true***), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : while (times <3) do p1 || 2 : ***while (count == 0) do p2*** || 3 : while (true) do p2>;

S9: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : while (times <3) do p1 || 2 : lock(m0); p2 || 3 : ***lock(m0); p2*** >;

S10: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***input(x) ; p1*** || 2 : while (true) do p2|| 3 : lock(m0); p2 >;

S11: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(m0, false), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : while (times <3) do p1 || 2 : ***lock(m0); p2*** || 3 : lock(m0); p2 >;

S12: < I = <1, 2, 3>, E = {(x = 0), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(***m0, true***), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : while (times <3) do p1 || 2 : while (true) do p2|| 3 : ***while (count == 0) do p2*** >;

S13: < I = < 2, 3>, E = {(x = 1), (buffer0 = 0), (buffer1 = 0), (count = 0), (times = 0) }, L = {(***m0, true***), {(m1, false), {(m2, false)}, Q = {(m0,<>), (m1, <>), (m2,<>)}, 1 : ***while (count == 2) do p1***|| 2 : while (true) do p2 || 3 : while (true) do p2>;

Using either *depth-first* or *width-first* traversal strategy, and continuously applying the structural rules, the *LTS* will eventually be constructed. The resulting LTS will be a

format of graph that consists of nodes that denote the states and edges that denote the transitions.

Moreover, such LTS needs to be further simplified by reducing the τ-transitions. The techniques to fulfill the task of simplification will be discussed in detail along with the implementation of a *path constraints* generation tool in next Chapter.

Chapter 5 Design and Implementation Detail

As mentioned in the previous chapters, the problem we consider in this thesis is to generate significant sets of *path constraints* automatically with a given *test case* and a design abstract. In this chapter, design and implementation details of such a tool will be presented. This tool is implemented in Java programming language with approximately four thousand lines.

5.1 Fundamental architecture of design

First of all, it is considered that the testing in this thesis is the *reproducible testing* based on the specifications of underlying concurrent program systems. Such specifications will be expressed by applying a formal specification language which is based on *process algebra*.

Second, with a given design abstract in terms of *process terms* in the specification, a *Labeled Transition System* can be derived by applying a number of structural rules which are defined in ***Chapter 3.5***. Such a generation of *LTS* will be accomplished automatically by the implementation tool that will be discussed in this chapter. Meanwhile, this tool provides a certain mechanism to remove the duplicated states while constructing such a *LTS*.

Third, this tool also supports the further simplification of the constructed *LTS*. By performing such a simplification, those internal transitions which actually have nothing to do with the generation of *path constraints* can be ignored. In particular, this simplification will be based on the determinization and minimization algorithms in the theories of automata.

Finally, with the simplified LTS, which is also known as the *control model*, a variety of significant sets of executing *path constraints* which are denoted by the labels can be generated. Usually, there are some criteria, such as *path coverage criterion*, *state coverage criterion* and *edge coverage criterion*, available for such a generation. The state

coverage guarantees that each state in the *control model* will be covered at least once, whereas the *edge coverage* ensures that each edge in the *control model* will be covered at least once. The *path coverage* is a much stronger criterion which requires each possible path is covered. Since the succeeding state yielded by a certain action from one specific state is not able to be determined in the *control model*, *state coverage* cannot be used here. In this case, the *edge coverage* criterion or the *path coverage* criterion may be considered for the *path constraints* generation.

5.2 Class Diagrams in the implementation tool

Major classes in this implementation tool include *StateItem*, *State*, *Label*, *Process*, *Variable* and *Monitor*.

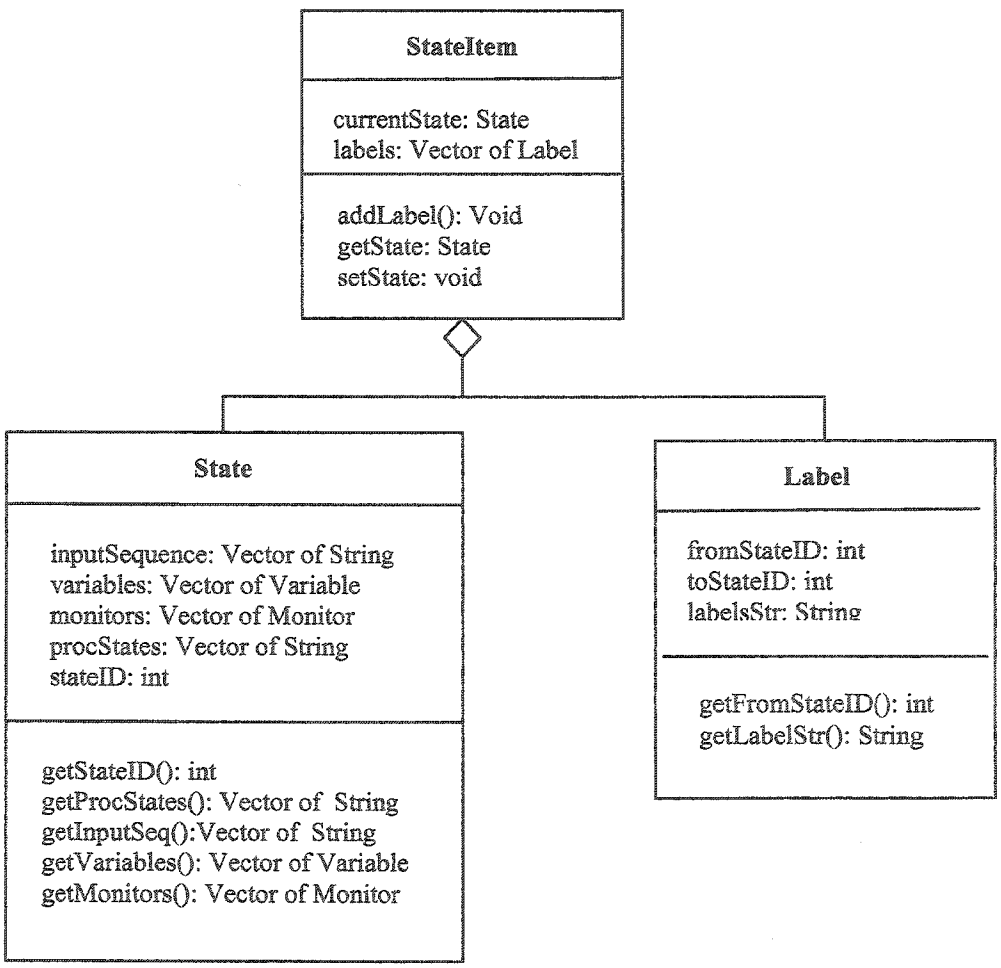

Figure 11: Class diagrams (I) for the implementation of *path constraints* generating tool

As shown in **Figure 11**,

- Class *Label* is used to describe the labels in the *LTS* and is made of an attribute *fromStateID* that denotes the identifier of its start state, an attribute *toStateID* that denotes the identifier of its end state, and an attribute *labelString* that describes the action that stimulates such a transition.
- Class *State* is used to describe the states in the *LTS* and consists of an attribute *stateID* that uniquely identifies a particular state; an attribute *inputSequence* that contains a sequence of input data; an attribute *variables* which is a set of instances of class *Variable* and contains information of all the variables that are used in the *PUT* along with their corresponding values; an attribute *monitors* that is a set of instances of class *Monitor* and contains information of all the monitors such as their lock status and contents of their waiting queues; and, an attribute *procStates* that describes the current status of each process, in particular, the current statement that each process is going to execute.
- Class *StateItem* is constructed by an instance of class *State* and a set of instances of class *Labels* that start from this state.

Also shown in **Figure 12** are the class diagram of the main components of class *State*-- classes *Variable* and *Monitor*.

- Class *Variable* is made of an attribute *varName* that denotes the name of this variable; an attribute *varType* that denotes the data type of this variable; an attribute *varSize* that denotes the size of this variable; and, an attribute *value* that denotes the value of this variable. For simplicity, the types of a variable considered in this thesis are limited to *integer*, *Boolean*, and *String*, even though this implementation tool is also able to handle the data type of *integer array* and *String array*. An instance of class *Variable* is employed to describe a variable used in the *PUT*.
- Class *Monitor* consists of an attribute *mid* that denotes the identifier of the monitor; an attribute *mstatus* that denotes the lock status of this monitor; and, an attribute *waitingQueue* that contains a set of identifiers of processes that are

currently waiting for the lock of this monitor. An instance of class *Monitor* is employed to describe a monitor used in the *PUT*.

Apart from class *Variable* and class *Monitor*, class *Process* is also described in **Figure 12.** Since the attribute *procStates* of class *State* contains the status of processes and the instance of class *Process* describes a process, class *Process* can be considered to be used by class *State*. Class *Process* contains an attribute *processID* that uniquely identifies a process, and an attribute *statements* that contains the design abstract of a process.

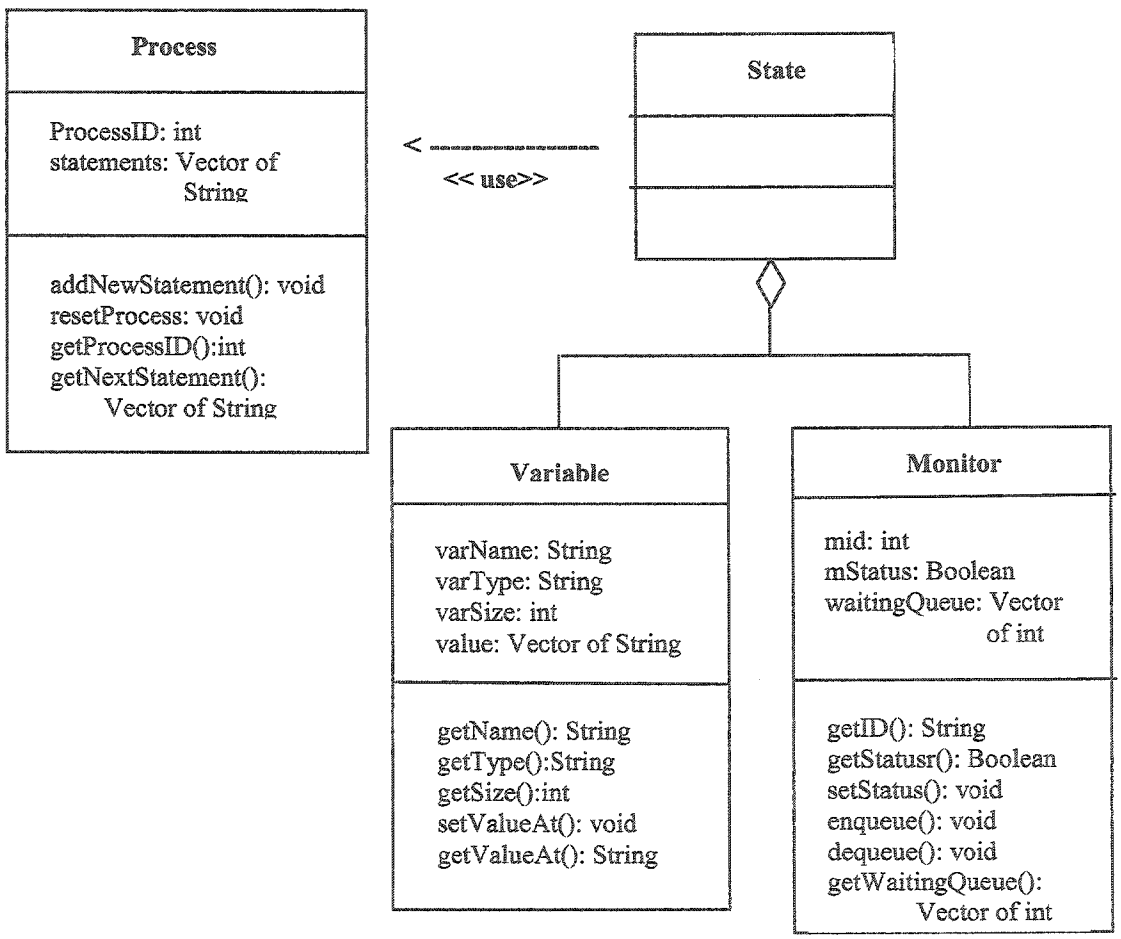

Figure 12: Class diagrams (II) for the implementation of *path constraints* generating tool

Besides such attributes defined for these classes, a set of methods is also designed for modifying or retrieving information on these classes. For instance, the methods *addLabel, getState* and *setState* on class *StateItem* are designed for inserting a new label to the instance of class *StateItem*, retrieving the state information, and modifying the state information of the instance of class *StateItem*, respectively.

5.3 Algorithms used in the implementation tool

According to the design architecture introduced in ***Section 5.1*** and based on the class diagrams discussed in the last section, three algorithms are used for accomplishing the *path constraints* generation. In particular, the algorithm for the *LTS* construction will be presented in ***Section 5.3.1***, the algorithm for producing the *control model* will be expressed in ***Section 5.3.2***, and the algorithm for the *path constraints* deriving will be discussed in ***Section 5.3.3***.

5.3.1 LTS generation

The fundamental part of the *path constraints* generating tool is the *LTS* generation. The process of *LTS* generation is actually the process of analyzing the *PUT* in terms of the design abstract. By applying a set of structural semantics rules, the behavior of the *PUT* will be simulated. The algorithm used in this tool for producing *LTS* is presented in **Figure 13**.

Algorithm for generating LTS:

Public Variables:
statesOfSystem;

generating(INPUT: *stateItem*)
BEGIN:
<Step1>
Retrieve the current state *thisState* from *stateItem;*
Retrieve the information of the input sequence, the current variables, the current monitors, and the current statuses of all the processes to *inputSequence, variables, monitors,* and *procStates* from *thisState,* respectively;
<Step2>

```
For each item of procStates do
    //such an item describes the status of a process
    Make copies of the state information into inputSequence_next,
    variables_next, monitors_next, and procStates_next for the succeeding
    state, respectively;
    Set flag isDone to false,
            searchNext to true,
    Set current label string thisLabelStr to empty;
    Copy this item to thisProcState;
    Retrieve the current process identifier thisProcID from thisProcState;
    Retrieve the current statement from thisProcState to currenStatement;
    If currentStatement is a stop statement then
        Remove the item corresponds to the current item from
        procStates_next;
        If procStates_next contains no element then
            Assign "stop" to thisLabelStr;
        Else
            Assign τ to thisLabelStr;
        EndIf
        Set isDone to "true";
    Else if currentStatement is an assignment statement then //":="
        Modify variables_next by evaluating the value of the expression at the
        right hand side of ":=" and assigning this value to the variable at the
        left hand side of ":=";
        Assign τ to thisLabelStr;
    Else if currentStatement is a conditional statement then //"if then else"
        Set a point for succeeding statement in procStates_next by evaluating
        the condition; //to denote the branch of "then" or" else" will be executed
        Assign τ to thisLabelStr;
    Else if currentStatement is a while-loop statement then  //"while do"
        Set a point for succeeding statement in the procStates_next by
        evaluating the condition; //indicate the body of while-loop or statements
                                  after while-loop will be executed
        Assign τ to thisLabelStr;
    Else if currentStatement is a input statement then //"input ()"
        Assign "(" + thisProcID + ",input)" to thisLabelStr;
        Assign first value of inputSequence to the variable indicated by input
        statement in variables_next;
        Remove the corresponding first data from inputSequence_next;
    Else if currentStatement is a lock statement then //"lock ()"
        Retrieve the monitor identifier from currentStatement to mid;
        Examine the lock status of monitor mid in monitors;
        If the lock is occupied by another process then
            Skip the remaining statements and proceed with next item;
        Else
            Set the lock status of the corresponding monitor in monitors_next
            to "ture";
            Assign "(" + thisProcID + ",lock, " + mid+")" to thisLabelStr;
        Endif
    Else if currentStatement is a waiting statement then //"waiting ()"
```

Skip the remaining statements and proceed with next item;
Else if *currentStatement* is *a lock_restart* statement then //"*lock_restart ()*"
Retrieve the monitor identifier form *currentStatement* to *mid*;
Examine the lock status of monitor *mid* in *monitors;*
If the lock is occupied by another process then
Skip the remaining statements and proceed with next item;
Else
Set the lock status of the corresponding *monitor* in *monitors_next* to *"ture";*
Assign τ to *thisLabelStr;*
Endif
Else if *currentStatement* is *a lock_end* statement then //"*lock_end ()*"
Retrieve the monitor identifier form *currentStatement* to *mid*;
Set the lock status of the corresponding monitor in *monitors_next* to *"false";*
Assign τ to *thisLabelStr;*
Else if *currentStatement* is *a wait* statement then //"*wait()*"
Retrieve the leaving monitor identifier and the waiting monitor identifier form *currentStatement* to *mid1* and *mid2,* respectively;
Assign "(" + *thisProcID* + ",wait, " + *mid2*+")" to *thisLabelStr;*
Set the lock status of the corresponding monitor for monitor *mid1* in *monitors_next* to *"false";*
Add the current process identifier *thisProcID into the waiting queue of the* monitor *mid2* in *monitors_next;*
Modify the succeeding statement of current process in the *procStates_next* to a *waiting* statement;
Set the value of searchNext to "false";
Else if *currentStatement* is *a notify* statement then //"*notify ()*"
Retrieve the monitor identifier form *currentStatement* to *mid*;
Assign "(" + *thisProcID* + ",notify, " + *mid*+")" to *thisLabelStr;*
Retrieve the first process identifier *pid* in the waiting queue of the monitor *mid* in *monitors*;
If the current statement of the process *pid* is a *waiting* statement and such a process is waiting in the waiting queue of the monitor *mid* then
Set the succeeding statement of the process *pid* to a *lock_restart* statement in *procStates_next;*
Remove *pid* from the waiting queue of the monitor that corresponds to the monitor *mid* in *monitors_next;*
Endif
Else if *currentStatement* is *a notifyAll* statement then //"*notifyAll ()*"
Retrieve the monitor identifier form *currentStatement* to *mid*;
Assign "(" + *thisProcID* + ",notifyAll, " + *mid*+")" to *thisLabelStr;*
For each process identifier *pid* in the waiting queue of the monitor *mid* in *monitors* do
If the current statement of the process *pid* is a *waiting* statement AND such a process is waiting in the waiting queue of the monitor *mid* then
Set the succeeding statement of the process *pid* to a *lock_restart* statement in *procStates_next;*
Remove *pid* from the waiting queue of the monitor that

```
                        corresponds to the monitor mid in monitors_next;
                    Endif
                Endfor
            Endif
    <Step3>
            If isDone is false AND searchNext is true then
                Retrieve the succeeding statement of the current process to modify
                procStates_next;
            Endif
    <Step4>
            Composite the succeeding state nextState by inputSequence_next,
            variables_next, monitors_next, and procStates_next;
            Generate a new label thisLabel by the state identifier of thisState, state
            identifier of nextState, and thisLabelStr;
            If there exists a state identical with the nextState then
                Modify the toStateID of thisLabel to the state identifier of this state;
            Else
                Generate a new stateItem nextStateItem by nextState;
                Add nextStateItem to statesOfSystem;
                Generating (nextStateItem);
            Endif
            Add thisLabel to stateItem;
        Endfor
    <Step5>
        Return;
END
```

Figure 13: Algorithm for generating *LTS*

In **Figure 13**, the variable *statesOfSystem* will contain all the generated instances of class *StateItem* along with the *LTS* generating, and is maintained publicly which means it is defined out of the scope of the method ***generating*** and will continue to exist after ***generating*** terminates. In the first step, the information of the current state, including input sequence, variables, monitors, and current status of each process, is retrieved from the input *stateItem* that is an instance of class *StateItem*. In the second step, a copy of such information is made for the purpose of generating the succeeding state. Meanwhile, a couple of binary flags are initialized: *isDone* that denotes if the current analyzing process terminates, and *searchNext* that denotes whether or not to retrieve the succeeding statement for the current process. As one important aspect of the state information, the current status of each process in terms of the current statement that is about to be executed is described by the variable *procStates*. The structural rules are applied

according to these statements in the second step. In fact, each item in *procStates* expresses not only the current statement of one specific process but also a pointer that is used to search the succeeding statement. When the current statement is a *lock* statement or a *lock_restart* statement, and in case the lock of the monitor, for which such a statement is requesting, is currently occupied by another process, the strategy used by this algorithm is to skip analyzing the current process and consider other processes first. The same strategy is also used when the current statement of the analyzed process is a *waiting* statement. Another special strategy, which generates the succeeding state by artificially inserting a *waiting* statement instead of searching the succeeding statement of the current process, will be applied when the current statement is a *wait* statement. The action of searching the succeeding statement is performed in the third step of this algorithm. With the succeeding statement of the current process, the succeeding status of variables in *variables_next,* the succeeding status of monitors in *monitors_next*, and the succeeding content of the input sequence in *inputSequence_next*, the succeeding state of the system can be generated in the fourth step. Meanwhile, according to the label string described by *thisLabelStr* and the identifier of the current state and the new generated state, a new label that expresses the evolution motivated by the current statement can be produced and added to *stateItem*. Finally, this algorithm generates a new instance of class *StateItem nextStateItem* by the generated instance of class *State nextState* while there exists no duplicate to this new state, and recursively invoke the method ***generating*** with the parameter *nextStateItem*.

Basically, following such an algorithm, the included valid statement may lead the system into different states, and further expansion can be carried out according to these states. Therefore, a critical problem of the state explosion will occur sooner or later. To deal with such a problem, one strategy that considers ignoring all the irrelevant states and labels is described in the algorithm in ***Section 5.3.2***. Another scheme considered in this algorithm also provides a big help to alleviate the state explosion. Such a scheme is based on the fact that it is most likely that a state led by performing a statement already existed in the *LTS*. In particular, two states can be considered as the same state in this thesis only if the current statement of each process, the values of all the variables, the status of all the monitors, and the content of the input sequence of these two states are exactly same. In

this case, the expansion will not be performed according to this statement execution since the expansion on the same state must have been performed and all the post states of the current state can also be approached from the previous same state. The only thing needs to do under such a circumstance according to this algorithm is to produce a new label which denotes this evolution to that existed state.

5.3.2 Simplifying LTS to the control model

By performing the algorithm presented in the last section, an *LTS* can eventually be generated. Since the scheme for excluding the duplicated states has been taken into account of such an algorithm, the generated *LTS* is actually a graph instead of a tree. However, due to the fact that the generated *LTS* still contains a large number of irrelevant states and labels for deriving the *path constraints*, certain simplifications of the *LTS* have to be done. The algorithm provided in **Figure 14** aims to perform such a task.

Algorithm for constructing Control Model:

```
Public Variables:
statesOfSystem, controlModel;

produceControlModel()
BEGIN:
    <Step1>
        Initialize controlModel;
    <Step2>
        For each item in statesOfSystem do
            If this item contains the initial state of the LTS then
                Search for all the items in statesOfSystem that contain the
                states can be reached from this initial state via one or more
                labels which have label string τ (or τ-label);
                Put these states and the initial state together as a new item
                 in controlModel;
            Endif
        Endfor
        For each item in controlModel do
            Copy this item to thisNode;
            For each state in thisNode do
                Find the item in statesOfSystem which contains such a state;
                For each non-τ label in this item do
                    If there is no label in thisNode has the same label string
                    as this label then
                        Create a new item in controlModel to contain the
                        end state of this label and all the states can be reached
```

```
                    from this end state via one or more τ-labels;
                    Create and add to thisNode a new label with the identifier
                    of thisNode, the identifier of the new generated item, and
                    the label string of this label;
                Else
                    Find the item in controlModel that is pointed by the label
                    which has the same label string as this label;
                    Add the end state of this label and all the states can be
                    reached from this end state via one or more τ-labels to this
                    item;
                Endif
            Endfor
        Endfor
        For each label in thisNode do
            If the state set of another item in controlModel is identical with the
            state set in the item that is pointed by this label then
                Remove the item that is pointed by this label from controlModel;
                Modify toStateID of this label by the identifier of such an
                identical item;
            Endif
        Endfor
    Endfor
<Step3>
    Initialize equivTable in which each item that is corresponding
    to the equivalence relation of each pair of state sets in controlModel;
    Initialize pairLists;//contains pairs of state sets
    For each item in controlModel do
        For each of all other items in controlModel do
            Check equivTable;
            If these two items have not proved to be distinguishable yet then
                If the label numbers in both items are equal AND
                each label in one item has a corresponding label in the
                other item with same label string then
                    Add these two items as a pair to a new set in pairLists;
                    Continue to check the equivalence relation of all the
                    succeeding pairs of items;
                Else
                    Set the relation between these two items to distinguishable
                    in equivTable;
                    Find the set of state pairs that contains these two items in
                    pairLists, and set the relation between the two items of
                    each pair to distinguishable in equivTable;
                Endif
            Endif
        Endfor
    Endfor
    For each item in controlModel do
        Check equivTable;
        For each equivalent item of this item in the controlModel do
            Remove such an equivalent item from controlModel;
```

```
                Modify all the labels pointed to the equivalent item to
                point to this item;
            Endif
        Endfor
    <Step4>
        Return;
END
```

Figure 14: Algorithm for constructing Control Model

Essentially, the *LTS* can be considered as a sort of *finite state automata* (or *finite automata*). Therefore, most well-defined theories of *finite automata* are suitable to deal with the problems of *LTS*. Basically, there are two types of *finite automata*: *Deterministic Finite Automata (DFA)* and *Nondeterministic Finite Automata (NFA)* [23]. The major difference between *DFA* and *NFA* is that with an input, while a state must be moved to exactly one specific state in a *DFA,* the successor of a state can be a set of zero, one, or more states in an *NFA*. Since the focus of this thesis is concurrent system testing, the *LTS* generated by the algorithm discussed in the last section is indeed an *NFA*. For each *DFA*, an equivalent *DFA* that has minimum states can be found by grouping those states that are equivalent. On the other hand, a *DFA,* which can do whatever an *NFA* can do, can always be constructed from such a given *NFA*. The process of constructing a *DFA* from an *NFA* is called determinization. Due to the fact that there is no way except a process of exhaustive enumeration to find a minimum-state *NFA equivalent to a given NFA*, it is necessary to perform the process of determinization before minimizing the state in an *NFA*. Considering our problem of simplifying the *LTS,* the determinization and minimization for such an *LTS* have to be applied. In fact, the algorithm provided in **Figure 14** is based on the idea of automata determinization and minimization [23].

As shown in **Figure 14**, there are also two public variables used in this algorithm: variable *statesOfSystem,* defined the same way as in the *LTS* generating algorithm, and variable *controlModel* that contains the information of all significant sets of state identifiers and sets of labels. In the second step of method **produceControlModel,** the *LTS* stored in *statesOfSystem* is *determinized* and stored in *controlModel.* Such a determinization involves constructing all subsets of the set of states in the *LTS*. First of

all, a subset that contains the initial state of the *LTS* and all states that can be reached from this initial state via one or more τ-labeled transitions will be generated. Second, from these states and according to each possible visible action, a variety of new subsets, each of which contains the succeeding states of these states via one specific non-τ label and all states that are reachable from these succeeding states via one or more τ-transitions, can be constructed. Third, we need to continuously construct such subsets from the existing state sets until all new constructed subsets are identical with other existing subsets or the states in the subset have no succeeding state. Finally, the determinized *LTS* has been generated in *controlModel* by considering each of these constructed state subsets as a new state. So far, all the τ-transitions in the *LTS* have also been eliminated. With such a determinized *LTS*, the process of minimization will be accomplished in the third step of method **produceControlModel.** In essence, this process is performed by grouping those states that are equivalent in the above-determinized *LTS*. The equivalence relation considered here is *trace equivalence*. According to the definition of *trace equivalence*, each pair of states in this determinized *LTS* will be examined. Two states are marked distinguishable, if:

- exactly one of these states is the final state that has no outgoing label;
- one state can be moved to its succeeding state via a transition on one specific action, while the other state cannot;
- the succeeding state pair of these two states via the transitions on corresponding action are found distinguishable.

The equivalence relation between any two states will be registered in a table that is described in the variable *equivTable*. Actually, only state distinguishablities can be determined by this algorithm. However, according to the theorem proved in [23], two states are indeed equivalent if these states are not distinguished by such a state distinguishing process. With the equivalence relation table, states in the determinized *LTS* can be partitioned into different groups, so that all states in the same block are equivalent and no pair of states from different groups is equivalent. Again, by considering each group of states as a new state, a minimized and determinized *LTS* that is also known as the *control model* is constructed.

5.3.3 Deriving the Path Constraints

With the *control model* constructed by the algorithm discussed in the last section, it is relatively simple to derive all significant sets of *path constraints*. Since the criterion of *state coverage* is not appropriate to be used in this case, as mentioned earlier, the *path coverage* criterion or the *edge coverage* criterion may be considered. The algorithm using the *path coverage* criterion for deriving the *path constraints* is provided in **Figure 15**, and the algorithm using the *edge coverage* criterion is presented in **Figure 16**.

```
Algorithm using path coverage for deriving the Path Constraints:
Public variables:
controlModel, pathConstraints;

generatePathConstraints1()
BEGIN
    <Step1>
        Initialize pathConstraints;
    <Step2>
        For each label from the initial state of controlModel do
            Copy this label to thisLabel;
            Copy the label string of thisLabel to variable thisPathConstraint;
            //contains one set of path constraints
            findNextContraint1(thisLabel, thisPathConstraint);
        Endfor
END


findNextContraint1(INPUT: thisLabel, thisPathConstraint)
BEGIN
    <Step1>
        Search the item that contains this state pointed by thisLabel in controlModel;
    <Step2>
        If no label form this state then
            Add thisPathConstraint to pathConstraints;
        Else
            For each label from this state do
                Copy this label to currentLabel;
                Copy all the items in thisPathConstraint to a new variable
                nextPathConstraint;
                Copy the label string of currentLabel to nextPathConstraint;
                findNextContraint1(currentLabel,nextPathConstraint );
            Endfor
        Endif
    <Step3>
```

```
        Return;
END
```

Figure 15: Algorithm using *path coverage* for deriving the *Path Constraints*

In the algorithm shown in **Figure 15,** one public variable *controlModel* is defined as in the *control model* constructing algorithm, and another public variable *pathConstraints* is introduced to contain all sets of path constrains derived from the *controlModel*. In the method **generatePathConstraints1,** the labels that start from the initial state are used to perform another method **findNextContraint1** in order to generate different sets of *path constraints*. The end state of each label will be examined in the method **findNextContraint1**, and such a method will be recursively invoked to process the labels from this end state until no label goes out from the current state. Eventually, all possible sets of *path constraints* can be registered in the variable *pathConstraints*.

```
Algorithm using edge coverage for deriving the Path Constraints:
Public variables:
controlModel, pathConstraints, hasNewLabel;

generatePathConstraints2()
BEGIN
    <Step1>
        Initialize pathConstraints;
    <Step2>
        For each label from the initial state of controlModel do
            If this label has not been covered yet then
                Mark this label to be covered;
                Set hasNewLabel to be true;
                Copy this label to thisLabel;
                Copy the label string of thisLabel to variable thisPathConstraint;
                 //contains one set of path constraints
                findNextContraint2(thisLabel, thisPathConstraint);
            Endif
        Endfor
END


findNextContraint2(INPUT: thisLabel, thisPathConstraint)
BEGIN
    <Step1>
        Search the item that contains this state pointed by thisLabel in controlModel;
     <Step2>
```

```
        If no label form this state then
            Set hasNewLabel to be false;
            Add thisPathConstraint to pathConstraints;
        Else
            For each label from this state do
                If this label has not been covered yet OR variable hasNewLabel
                is true then
                    Mark this label to be covered;
                    Set hasNewLabel to be true;
                    Copy this label to currentLabel;
                    Copy all the items in thisPathConstraint to a new variable
                    nextPathConstraint;
                    Copy the label string of currentLabel to nextPathConstraint;
                    findNextContraint2(currentLabel,nextPathConstraint );
                Endif
            Endfor
        Endif
    <Step3>
        Return;
END
```

Figure 16: Algorithm using *path coverage* for deriving the *Path Constraints*

One difference between the algorithms shown in **Figure 15** and **Figure 16** is that a new variable *hasNewLabel* is introduced in the latter to denote if the current path contains at least one uncovered edge. Meanwhile, a Boolean value will be associated with each label to denote whether or not this label has been covered at least once. Thus, each time a label will be added to construct a new set of path constraints while either this label has not been covered yet or such a set contains at least one uncovered label. As a consequence, the algorithm in **Figure 16** guarantees that each label in the control model must be covered at least once by the derived sets of *path constraints*.

Up to this point, the structure and the algorithms of the *path constraints* generating tool have been provided in detail. To illustrate the performance of this tool, the evaluation and some empirical results will be presented in ***Chapter 6***.

Chapter 6 Evaluation of the proposed framework

This Chapter discusses the evaluation of the proposed framework. This framework involves an implementation tool for deriving all significant sets of *path constraints* for *reproducible testing*. The algorithms used in such an implementation tool, which include generating the *LTS*, simplifying this *LTS*, and deriving *path constraints*, have been presented in ***Chapter 5***. In the first section of this Chapter, some computational issues of the *LTS* generation will be considered. Then, the empirical results of deriving *path constraints* with the *path coverage* criterion will be investigated in the second section. Finally, in ***Section 6.3***, the results of using the *edge coverage* criterion over a number of typical examples will be evaluated.

6.1 Computational issues

With the *LTS* generated by the algorithm presented in ***Chapter 5***, the behaviour of the *PUT* can be simulated. However, such an *LTS* suffers from the problem of *state explosion*. That is, the number of states in the *LTS* may increase exponentially as the number of processes in the PUT grows.

The number of states in the LTS is determined by a variety of factors. As mentioned in ***Chapter 3***, each state in the *LTS* consists of a sequence of input values, the mapping from variables to their values, the locking status of all monitors, the status of monitor waiting queues, and the status of each process. Specifically, the number of states in the *LTS* is in a proportional order to the number of input values, the number of possible values of each variable, the number of monitors for its locking mechanism, and the number of statements in each process. In addition, the number of states in the *LTS* is exponential to the number of processes due to the fact that the number of states of each monitor's waiting queue is the total number of permutations of zero or more processes taken from the set of processes.

Obviously, there will be a vast number of states in the *LTS* if any of the above factors gets bigger. However, such a bad situation is unlikely to happen in practice. Our

experiment actually provides some valuable data to show the scalability of our approach. In particular, according to our experiment, the functionality of the simplification provided by this tool considerably reduces the number of states in the generated *LTS*. We show in the following, the empirical results on this issue.

6.2 Empirical results of deriving path constraints with path coverage criterion

The *producer consumer problem* is a classical concurrent example and has been illustrated in ***Chapter 4***. Consider this example with one *producer* and two *consumer*s, and alter the times of producing new data by the *producer;* the empirical results by performing the implementation tool with the *path coverage* criterion are shown in **Table 1**.

Table 1 compares the number of states and labels in the *LTS* and in its corresponding *control model* and records the number of derived paths and executing time while the *producer* performs its task a different number of times. The line chart in **Figure 17** demonstrates the relation between the number of derived paths and times of producing the new data by the *producer*. According to the empirical results shown in **Table 1**, it is obvious that the number of states and labels in the *LTS* has been reduced significantly in the control model with this implementation tool. However, a major deficiency, which is that the number of path constraint sets increases exponentially while the number of labels is getting bigger, comes from the application of the *path coverage* criterion as shown in **Figure 17**. Therefore, it can be concluded that with the *path coverage* criterion, the scalability problem may not be handled properly by the implementation tool. Consequently, another strategy - edge coverage criterion - will be considered for the task of *path constraints* generation in this implementation tool, and the empirical results of deriving *path constraints* with the *edge coverage* criterion will be provided in the next section.

	In *LTS* (Before Simplifying)		In *Control Model* (After Simplifying)			
Producing times	State Number	Label Number	States Number	Label Number	Path Number	Time (Sec)
1	531	1108	33	49	42	2
2	1388	2911	93	159	1776	10
3	2959	6235	193	345	78602	137

Table 1: Empirical results of *producer consumer problem* with *path coverage* criterion

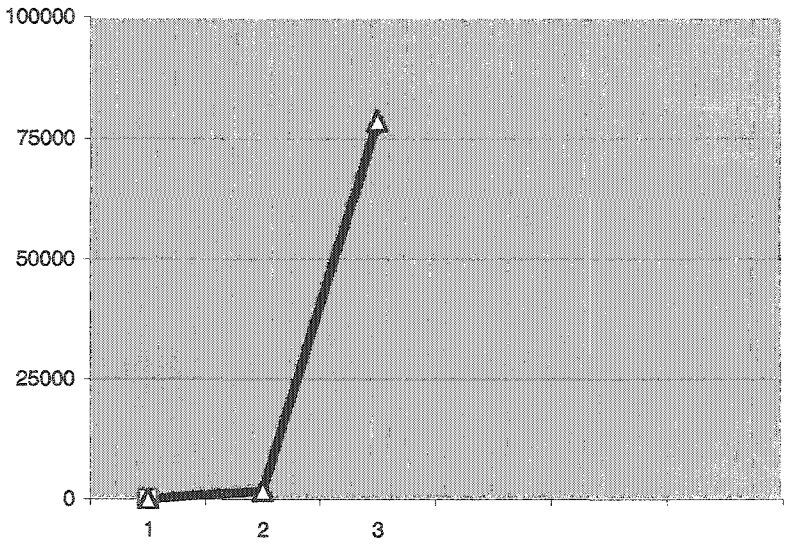

Figure 17: Line chart of increasing path number with *path coverage* criterion

6.3 Empirical results of deriving path constraints with edge coverage criterion

Evaluation of this framework is complex as there are various circumstances in the category of concurrent system testing. The experiments chosen to evaluate this framework should reflect the behaviour of the implementation tool in terms of significant reduction of state and label numbers from *LTS* to *control model,* the number of derived sets of *path constraints*, and the execution time. Evaluation of the above-mentioned testing method in experiments requires careful selection of typical examples that contain general scenarios in most concurrent systems. Hence, the *producer consumer problem*

will be considered again in ***Section 6.3.1,*** and two other typical examples: *Reader & Writer problem* and the *Sleeping Barber problem* will be illustrated in ***Section 6.3.2*** and ***Section 6.3.3***, respectively.

6.3.1 Reconsider producer consumer problem

To demonstrate the performance of the implementation tool by employing the *edge coverage* criterion, the example of *producer consumer problem* is considered again first.

Since nothing else but the strategy of deriving sets of *path constraints* from the *control model* has been changed, only the results of path number and executing time are different in **Table 2** from those in **Table 1**. It is not surprising that the number of derived paths no longer explodes while the numbers of state and label are increasing since the *edge coverage* criterion is applied. In fact, the increase of path numbers will be linear as the line chart shown in **Figure 18**. Also, comparing **Table 1** and **Table 2**, another major difference is that the growing rate of the execution time, along with the boost of valid number of states and labels by using the *edge coverage* criterion, is much lower than that obtained by using the *path coverage* criterion.

	In *LTS* (Before Simplifying)		In *Control Model* (After Simplifying)			
Producing times	State Number	Label Number	States Number	Label Number	Path Number	Time (Sec)
1	531	1108	33	49	18	1
2	1388	2911	93	159	68	7
3	2959	6235	193	345	154	38

Table 2: Empirical results of *producer consumer problem* with *edge coverage* criterion

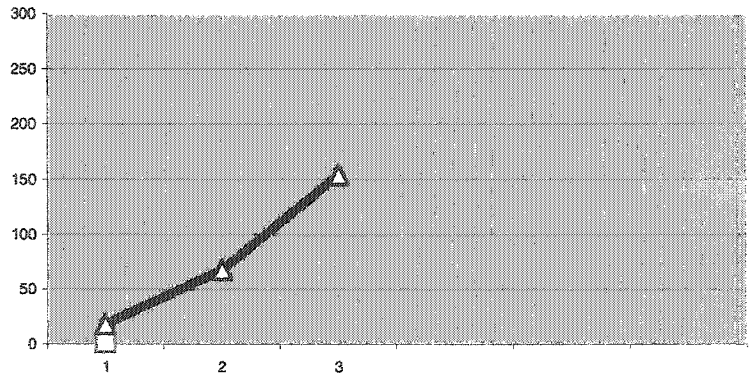

Figure 18: Line chart of increasing path number with *edge coverage* criterion

6.3.2 The example of Reader & Writer problem

Reader & Writer problem is somewhat similar to *producer consumer problem*, and is a typical concurrent example, too. In such a problem, there exists a shared database, which can be queried by the *reader* processes and be examined and altered by the *writer* processes. Due to the characteristics of the *reader* and *writer*, the database can be assessed concurrently by different *readers*, while *writers* require exclusive access of the database.

The design abstract for the solution of *Reader & Writer problem* is presented in **Figure 19**. Two integer variables *nr* and *nw* are used in this solution to denote the numbers of *readers* and *writers* that are currently processing the database, respectively. Before operating the database, each process enters a critical section protected by monitor *m1*. The initial value of all these variables will be 0. If there is no any other process that either a *reader* or a *writer* currently accessing the database, a *write* process is allowed to alter the database, and increase the variable *nw* by 1; otherwise, this *write* process puts itself in the waiting queue of monitor *m2*. However, a *reader* process is allowed to query the database and increase the variable *nr* by 1 unless there is a *writer* process that is operating the database. A reader waits in the waiting queue of monitor *m1* if the number of *writers nw* is larger than 0. After operating the database, each process enters another critical section that is also protected by monitor *m1*. Either number of *writers* or number of *readers* will be decreased by 1; and, the *write* process will awaken the first *writer*

process that waits on monitor *m2* and all *reader* processes that wait on the monitor *m1*, or the *reader* process will awaken the first *writer* process that waits on monitor *m2* if no more *readers* are accessing the database.

```
* Reader&Writer *
<variables> {{ nw:int },{ data:int },{ nr:int },{ dataOfReader :int }}
<monitors> {m0,m2,m1}

<process>:Writer
lock ( m0 ) {
      while ( nr > 0 || nw > 0 ) do {
         wait ( m0 , m2 );
      }
      nw := nw + 1;
}
lock_end ( m0 );
data := data + 1;
lock ( m0 ) {
      nw := nw - 1;
      notify ( m2 );
      notifyAll ( m1 );
}
lock_end ( m0 );
stop;
<process>:Reader
lock ( m0 ) {
     while ( nw > 0 ) do {
          wait ( m0 , m1 );
     }
     nr := nr + 1;
}
lock_end ( m0 );
dataOfReader1 := data;
lock ( m0 ) {
     nr := nr - 1;
     if ( nr == 0 ) then {
        notify ( m2 );
     }
}
lock_end ( m0 );
stop;
```

Figure 19: Design abstract for the example of *Reader & Writer*

The results of performing the implementation tool with this example of *Sleeping Reader & Writer problem* are displayed in **Table 3.**

Number	In *LTS* (Before Simplifying)		In *Control Model* (After Simplifying)			
Readers/ Writers	State Number	Label Number	States Number	Label Number	Path Number	Time (Sec)
2/1	779	1404	85	122	39	2
3/1	6443	14522	357	639	285	114
2/2	7013	15215	491	855	366	118

Table 3: Empirical results of *Reader & Writer problem*

6.3.3 The example of Sleeping Barber problem

Sleeping Barber problem is another classic synchronization problem, and also a representative of practical problems. The important client/server relationship that often exists between different processes is illustrated in this problem.

The situation described by *Sleeping Barber problem* is: There is a barber-shop in a small town. The shop has a barber, a barber chair, and a waiting room with several chairs. The barber spends his lifetime to serve customers, and sleeps in the barber's chair when none are in the shop. When a customer arrives and finds the barber is sleeping, the customer awakens the barber, and sleeps in the barber's chair while the barber cuts his hair. If the barber is busy when a customer arrives, the customer goes to sleep in the chair in the waiting room if at least one of such chairs is available; otherwise, the customer comes back later. After finished cutting, the barber awakens the customer who has received a haircut and lets him leave. If there are waiting customers, the barber then awakens one and gives another haircut; otherwise, the barber goes back to sleep until a new customer arrives.

```
* sleepingBarber *
<variables> {{ num_cust:int },{ total:int }}
<monitors> {m1,m2,m3}

<process>:Barber
while ( total<3 ) do {
     lock ( m1 ) {
          if ( num_cust == 0 ) then {
               wait ( m1 , m2 );
          } else {
          }
          num_cust := num_cust - 1;
          notify ( m3 );
          total := total + 1;
     }
     lock_end ( m1 );
     notifyAll ( m1 );
}
stop;
<process>:Customer
lock ( m1 ) {
     while ( num_cust == 2 ) do {
          wait ( m1 , m1 );
     }
     num_cust := num_cust + 1;
     if ( num_cust == 1 ) then {
          notify ( m2 );
     } else {
     }
     wait ( m1 , m3 );
}
lock_end ( m1 );
stop;
```

Figure 20: Design abstract for the example of *Sleeping Barber*

The design abstract for a solution of *Sleeping Barber problem* is given in **Figure 20**. In this solution, three monitors, *m1*, *m2*, and *m3*, are employed. In particular, monitor *m1* is used to ensure the mutual exclusion of the barber's cutting and the customer's entering the barber-shop; monitor m2 is used to signal the barber that a new customer arrives; and monitor *m3* is used to signal the customer that the barber has finished his cutting. Variable *total* is used to denote the total number of customs that have been served by the barber, and the initial value is 0. Besides, a variable *num_cust* denotes the available number of chairs in the barber's waiting room, and the value will be given with

an initial state of this system. If a customer gains the lock of monitor *m1* and finds no chair in the waiting room is available, he gives up the lock and waits in the waiting queue of monitor *m1*. On the other hand, if the barber gains the lock of monitor *m1* and finds no customer is waiting, he waits in the waiting queue of monitor *m2*; otherwise, he cuts one customer's hair and wakes up customers that wait on the monitor m1 if there is any. Since only static processes will be consider as mentioned earlier, the number of customer processes will be same as the number of customers that the barber is allowed to serve. In this case, it is not necessary for the customer process to examine if the barber process is still running due to the fact that each customer will be eventually served.

Number	In *LTS* (Before Simplifying)		In *Control Model* (After Simplifying)			
Barber/Customers	State Number	Label Number	States Number	Label Number	Path Number	Time (Sec)
1/2	375	642	49	63	16	1
1/3	3175	6334	183	265	84	37
1/4	27963	62278	792	1274	484	4714

Table 4: Empirical results of *Sleeping Barber problem*

The results of performing the implementation tool with this example of *Sleeping Barber problem* are listed in **Table 4.** According to this table, three points should be noticed:

- Although as the number of processes increases, the number of states and labels in the generated *LTS* rises remarkably, the numbers of states and labels in the *control model* that is simplified from the *LTS* by the implementation tool does not accumulate likewise;

- The number of derived sets of *path constraints* by applying the *edge coverage* criterion increases linearly while the number of processes grows;

- The execution time for deriving *path constraints* increases significantly as the number of processes increases.

Chapter 7 Conclusion

In this thesis, a framework to automatically generate all significant sets of *path constraints* for reproducible testing has been proposed. The purpose of these derived *path constraints* is to gain desired deterministic control over the non-deterministic testing environment.

Formal methods process terms and *Labelled Transition System (LTS)* are introduced to specify the design abstract of the *PUT* and to construct the model for simulating the behaviour of such a *PUT*, respectively. Due to the fact that the *LTS* is indeed a sort of Nondeterministic Finite Automata, the algorithms of determinization and minimization in theories of automata are applied to simplify the *LTS* to the desired *control model* by reducing those internal transitions according to *trace equivalence*. The *control model* contains a minimum number of states and labels that are necessary for generating the *path constraints*. Finally, the *edge coverage* criterion which guarantees that each label in *control model* will be covered at least once is used to derive all significant sets of *path constraints*.

The experiments presented in ***Chapter 6*** demonstrate that it is efficient and effective to construct the *control model* that is made up of only necessary information for deriving the *path constraints* by simplifying the *LTS* with this implementation tool. Further, with the *edge coverage* criterion, this implementation tool derives only significant sets of *path constraints*. Since the number of derived sets of *path constraints* is manageable and the increase of such a number is linear as the size of the control model grows, the scalability problem can be handled properly by this tool.

Bibliography

[1] A. Abdurazik, P. Ammann, D. Wei and J. Offutt. Evaluation of three specification-based testing criteria. Engineering of Complex Computer Systems, ICECCS 2000. In Proc. OF Sixth IEEE International Conference, pages: 179 -187, Sept. 2000.

[2] Gregory R. Andrews. Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley, Inc. 2000.

[3] P. C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. ACM Transactions on Computer Systems (TOCS), Volume 13 Issue, February. 1995.

[4] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, Protocol Specification, Testing, and Verification VIII, pages 63- 74, NorthHolland, 1988.

[5] A. Bechini and K.-C. Tai. Design of a toolset for dynamic analysis of concurrent Java programs. In Proc. of the 6th International Workshop on Program Comprehension, Italy, Pages: 190 -197, 24-26 June 1998.

[6] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In Lecture Notes in computer science Vol. 1067, pages 187-195, Springer-Verlag, 2001.

[7] R.H. Carver. Testing Abstract Distributed Programs and Their Implementations. J. System and Software, special issue on Software Engineering for Distributed Computing, pages. 223-237, June 1996.

[8] J. Chen. On using static analysis in distributed system testing. In proc, of the 2nd International Workshop on Engineering Distributed Objects (EDO 2000). Lecture Notes in Computer Science Vol. 1999, pages 145-162, Springer- Verlag, 2000.

[9] J. Chen. Working on Scenarios for Reproducible Testing. The 5th International Conference on Formal Engineering Methods (ICFEM 2003). Lecture Notes in Computer Science Vol 2885, pages 34-47, Springer-Verlag, 2003.

[10] X. Cai and J. Chen. Control of nondeterminism in testing distributed multi-threaded programs. In Proc. of the First Asia-Pacific Conference on Quality Software (APAQS 2000), pages 29-38. IEEE Computer Society Press, 2000.

[11] R.Carver and K.-C. Tai. Replay and testing for concurrent programs. IEEE Software, pages 66-74, Mar. 1991.

[12] R.Carver and K.-C. Tai. Static analysis of concurrent software for deriving synchronization constraints, In proc. Of IEEE International conference, Distributed Computer System, pages. 544-551, May 1991.

[13] R.Carver and K.-C. Tai. Use of sequencing constraints for specification-based testing of concurrent programs. IEEE Transitions on Software Engineering, 24(6):471-490, June 1998.

[14] S.K. Damodaran-Kamal and J.M. Francioni. Nondeterminancy: Testing and Debugging in Message Passing Parallel Programs. Proc. ACM/ONR Workshop Parallel and Distributed Debugging, ACM SIGPLAN Notices, vol. 28, no. 12, pages 118–128, Dec. 1993.

[15] R. A. DeMillo and A. J. Offutt, Constraint-Based Automatic Test Data Generation. IEEE Trans on Software Eng, Vol 17, No. 9, pages 900-91, Sept. 1991.

[16] R. A. DeMillo and A. J. Offutt, Experimental Results from an Automatic Test Case Generator. ACM Trans on Software Eng. and Method, Vol 2, No. 2, pages 109-127, April 1993.

[17] M.-C. Gaudel. Testing can be formal, too. In P.D. Mosses, M. Nielsen, and M.I. Schewartzbach, editors, TAPSOFT 95: Theory and Practice of Software Development, pages 82-96. Lecture Notes in Computer Science 915, Springer-Verlag, 1995.

[18] W. Geurts, K. Wijbrans, and J. Tretmans. Testing and formal methods—BOS project case study. In EuroSTAR'98: 6th European Int. Conference on Software Testing, Analysis & Review, pages 215-229, Munich, Germany, November 30- December 1 1998.

[19] A. Hall. Seven myths of formal methods. IEEE Software, 6(9):11-19, 1990.

[20] R. M. Hierons, "Testing from a Z Specification". *Software Testing, Verification and Reliability*, Vol. 7. pages 19-33, 1997.

[21] C. Hoare. A calculus of total correctness for communicating process. Science of Computer Programming, 1:49- 72, 1981

[22] G.J. Holzmann. Design and Validation of Computer Protocols. Prentice-Hall Inc., 1991.

[23] John E. Hopcroft, Rajeev Motwani, and Jeffery D. Ullman. Introduction to automata theory, language, and computation. Addison-Wesley, 2nd edition, 2001.

[24] D.J. Hatley and I.A. Pirbhai. Strategies for Real-Time System Specification. Dorset House, 1987.

[25] L. Heerink and J. Tretmans. Formal methods in conformance testing: A probabilistic refinement. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, Int. workshop on Testing of Communicating systems 9, pages 261-276. Chapman & Hall, 1997.

[26] G.H. Hwang, K.C. Tai, and T.L. Huang. Reachability Testing: An Approach to Testing Concurrent Software. Proc. Int'l J. Software Eng. and Knowledge Eng., vol. 5, no. 4, pages 493–510, Dec. 1995.

[27] ISO/IEO JTC1/SC21 WG7, ITU-T SG 10/Q.8. Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO-ITU-T, Geneve, 1996.

[28] E. Itoh, Z. Furukawa, and K. Ushijima. A prototype of a concurrent behaviour monitoring tool for testing concurrent programs. In proc of Asia- Pacific Software Engineering Conference (APSEC' 96), pages 345-354, 1996.

[29] P. Kars. Formal Methods in the Design of a Storm Surge Barrier Control System. In G. Rozenberg and F. W. Vaandrager, editors, Lectures on Embedded Systems, pages 353-367. Lecture Notes in Computer Science 1494, Springer-Verlag, 1998.

[30] D. Kung, N. Suchak, P. Hsia, Y. Toyoshima, and C. Chen. On object state testing. In proc. Of the 18th International computer software and applications conference: COMP-SAC '94, pages 222-227, IEEE Computer Society Press, 1994.

[31] B. Karacali and K. Tai. Automated test sequence generation using sequencing constraints for concurrent programs. In proc. Of the International Symposium on Software Engineering for Parallel and Distributed Systems, pages 97-106, IEEE Computer Society Press, 1999.

[32] T. Leblanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. IEEE Transitions on Computers, 36(4):471-482, Apr. 1987.

[33] N.A. Lynch and M. R. Tuttle. An introduction to Input/Output Automata. CWI Quarterly, 2(3):219-246,1989.

[34] D. Lee and M. Yannakakis. Principles and methods for testing finite state machines. The Preceedings of the IEEE, August 1996.

[35] R. Milner. Communication and Concurrency. Prentice Hall, London, 1989.

[36] N. Mittal and V. K. Garg. Debugging Distributed Programs Using Controlled Re-execution. In Proc. of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC-00), 2000.

[37] R. Nicola and M. Hennessy. Testing equivalence for processes. Theoretical Commputer Science, 34:83- 133, 1984.

[38] A. J. Offutt and S. Liu. Generating Test Data from SOFL Specifications. *Technical Report ISSE-TR-97-02. Department of Information and Software Systems Engineering, George Mason University, Fairfax VA*, pages 1-24, 1997. http://www.isse.gmu.edu/faculty/ofut/rsrch/papers/soflsp.ps

[39] J.M. Spivey. The Z Notation: a Reference Manual (2nd edition). Prentice Hall, 1992.

[40] P. Stocks and D. Carrington. A framework for specification-based testing. Software Engineering, IEEE Transactions, Volume: 22 Issue: 11, pages: 777 -793 Nov. 1996.

[41] H. Sohn, D. Kung and P. Hsia. State-based reproducible testing for CORBA applications. In Proc. of IEEE International Symposium on software Engineering for Parallel and Distributed Systems (PDSE'99), pages 24-35, LA, USA, May 1999.

[42] H. Sohn, D. Kung, P. Hsia. Y. Toyoshima, and C. Chen. Reproducible testing for distributed programs. In Proc. of the 4th International Conference on Telecommunication Systems, Modeling and Analysis, pages 172-179, Nashiville, Tennessee, Mar. 1996.

[43] H. Sohn, D. Kung, P. Hsia, Y. Toyoshima and C. Chen. Reproducible Testing of Distributed and Concurrent Programs. 12th International Society for Computers and Their Applications, Sanfransisco, CA, March 7 - 10, 1996.

[44] K.C. Tai. Reachability Testing of Asynchronous Message-Passing Programs. Proc. IEEE Int'l Workshop Software Eng. for Parallel and Distributed Systems, pages 50–61, May 1997.

[45] J. Tretmans. A Formal Approach to Conformance Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[46] J. Tretmans. Testing labelled transition systems with inputs and outputs. In A. Cavalli and S. Budkowski, editors, *Participants Proceedings of the Int. Workshop on Protocol Test Systems VIII -- COST 247 Session*, pages 461-476, Evry, France, September 4-6 1995.

[47] J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. Computer Networks and ISDN Systems, Vol. 29, pages: 49 – 79, 1996

[48] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. Software- concepts and tools, 17(3): 103-120, 1996.

[49] J. Tretmans. Testing transition system: A Formal Approach. In proc. Of the 10th International Conference on concurrency theory. Lecture Notes in Computer Science Vol. 1664, pages 46-65, Springer-Verlag, 1999.

[50] J. Tretmans. Specification Based Testing with Formal Methods: From Theory via Tools to Applications. In A. Fantechi, editor, *FORTE / PSTV 2000 Tutorial Notes*, Pisa, Italy, October 10 2000.

[51] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7th European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8-12, 1999. EuroStar Conferences, Galway, Ireland.

[52] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural Testing of Concurrent Programs. IEEE Trans. Software Eng., vol. 18, no. 3, pages 206–215, Mar. 1992.

[53] Q. M. Tan, A. Petrenko, G. V. Bochmann and G. Luo. Testing Trace Equivalence for Labeled Transition Systems. Département IRO, Université de Montréal, May 1995.

[54] Q.M. Tan, A, Petrenko, G.v.Bochmann. Checking experiments with labeled transition systems for trace equivalence. In the proc of IFIP 10th International Workshop on Testing of Commnication Systems (IWTCS'97), Korea, 1997.

[55] Q.M. Tan, A, Petrenko, G.v.Bochmann. Deriving specifications in the form of labeled transition systems. Département IRO, Université de Montréal, June 1997.

[56] T. H. Tse and Z. Xu. Test Case Generation for Class-Level Object-Oriented Testing. Quality Process Convergence: Proceedings of 9th International Software Quality Week (QW '96), San Francisco, California, pages 4T4.0-4T4.12 (1996). http://www.cs.hku.hk/~tse/Papers/xqwps.zip.

[57] R.D. Yang and C.G. Chung. Path Analysis Testing of Concurrent Programs. Information and Software Technology, vol. 34. no. 1, pages 43–56, Jan. 1992.

VITA AUCTORIS

NAME:	Jun Li
PLACE OF BIRTH:	Shanghai, China
YEAR OF BIRTH:	1972
WORKING EXPERIENCE:	Shanghai Waigaoqiao Free Trade Zone Xin Developing Co., Ltd, Shanghai, China 1994- 1996 Programmer 1997- 1999 System Analyst
	Shanghai Dong Lian software Co., Ltd, Shanghai, China 2000- 2001 System Analyst
EDUCATION:	ChengZhong High School, Shanghai, China 1984-1990
	University of Shanghai (Former named University of Shanghai Science and Technology), Shanghai, China 1990- 1994 B.Sc.
	University of Windsor, Windsor, Ontario, Canada 2001- 2004 M.Sc.